2004

Autonomous Robotic Automation Systemwith Vision Feedback

Jeffery Rosino
University of Central Florida

AUTONOMOUS ROBOTIC AUTOMATION SYSTEM WITH VISION FEEDBACK

by

JEFFERY M ROSINO
B.S. University of Central Florida, 2000

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Electrical and Computer Engineering
in the College of Engineering & Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2004

ABSTRACT

In this thesis, a full design, development and application of an autonomous robotic automation system using vision feedback is performed. To realize this system, a cylindrical manipulator configuration is implemented, using a personal computer (PC) based PID controller from National Instruments. Full autonomous control will be achieved via a programmable human machine interface (HMI) developed on a PC using Borland C++ Builder. The vision feedback position control is accomplished using an ordinary "off-the-shelf" web camera.

The manuscript is organized as follows; After Chapter 1, an introduction to automation history and its role in the manufacturing industry, Chapter 2 discusses and outlines the development of the robotic kinematics and dynamics of the system. A control strategy is also developed and simulated in this chapter. Chapter 3 discusses color image processing and shows the development of the algorithm used for the vision feedback position control. Chapter 4 outlines the system development, which includes the hardware and software. Chapter 5 concludes with a summary, and improvement section.

The process used as a basis for the design and development of this thesis of this thesis topic was constructed from a manual capacitor orientation check test station. A more detailed definition and objective is presented in the introduction.

To my wife, my son, and my parents.

ACKNOWLEDGMENTS

I want to thank Hermes Norero, Tito Visi, and Dave Thompson for taking time to help me in the development of the system. Hermes provided a true 3-dimentional model of the cylindrical manipulator; Tito helped with the gripper attachment assembly; and Dave provided me with his insight and knowledge of image processing.

I want to give special thanks to my advisor, Dr. Zhihua Qu, who continued to advise and support me semester after semester.

Finally, I want to thank my wife, Angela, for her love, encouragement, and devotion. Without her support and sacrifices, I would not have completed my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Degree-Of-Freedom.....DOF

Denavit and Hartenburg.....D-H

Center of Gravity.....CG

Symmetric Positive Definite.....SPD

Proportional-Derivative.....PD

Proportional-Derivative-Integral.....PID

1 INTRODUCTION

Technology and science are continuously advancing society. The rapid progress of technical advances has made it difficult for our political, educational, and religious institutions, as they struggle to digest the changes brought about by these advancements. Many are apprehensive of what the future may hold, but there is no doubt that the human race has the great ability to adapt and embrace these changes as they may make our lives easier, safer, and more efficient.

The present age has been given many names. This century alone has been dubbed, the nuclear age, space age, computer age, and the automation age. The later is the focus of this manuscript. In the field of manufacturing, the human race is at the threshold of a second Industrial Revolution, a revolution in Automation.

The word automation, which is short for "Automatic Motivation," was coined in the 1940's at the Ford Motor Company, and was used to describe the collective operation of many interconnected machines [5]. Automation today is defined in most standard dictionaries as "The automatic operation or

control of equipment, a process, or a system". A particular industry that has and still is being revolutionized by automation is manufacturing.

Manufacturing has benefited greatly from the automation industry. The greatest factor contributing to the full out automation of processes is due to human safety concerns. Today's society has taken great leaps to insure the well being and safety of the people. Organizations have devoted their entire existence to creating standards, and to uphold these standards, such as the Occupational Safety & Health Administration (OSHA). Automation has enabled many industries to still produce products that are developed in hazardous and unhealthy locations without the health or safety of its employee's being compromised. By far the biggest industry affected is the automobile industry. Industries that utilize the assembly line style of production are greatly enhanced by automation as well. Automation has enabled factories to reduce production costs, increase the quality of the products, and improve productivity.

In today's modern society, automation and robots go hand-in-hand. Robots represent the highest form of automation [5]. Robotics has become an industry in itself. The robotic industry deals with the design, control, programming, manufacture, and

application of a robot [4]. The term robot comes from the Czech and means “forced labor”. Webster’s dictionary defines the term; “as a machine in the form of a human being that performs mechanical functions of a human being” [3]. Today’s robots look nothing like a human, but perform the work of humans were the safety of a human is at risk.

A typical fully automated robot will consist of many mechanical parts (manipulator), sensors, and a sophisticated control system. Modern automation trends have lead to the implementation of vision systems integrated into a robotic system. Cameras and vision systems have become standard automation components [6]. Robotic vision systems are very useful for product inspection, part presentation, monitoring of assembly accuracy and component location.

A process that will be fully automated and part of the automation requires the use of a robotic system, needs the resources from a multi-disciplinary organization or an individual who is a robotics engineer. A robotics engineer should have knowledge in the fields of mechanical, electrical, computer, and application engineering, with emphasis on electrical engineering. If a vision system is to be integrated into the system, the designer should have knowledge in the area of image processing.

The following chapters go through the design, and development of a system that is to be automated using a vision based robotic system. The design phase is broken up into two chapters; the first being the robot design which consists of the manipulator and control design, and the second focuses on the image processing design and integration of the vision into the system. The design and development phase is shown in chapter 5, which consists of the hardware and software development and integration.

For this manuscript, the process to be automated is a product inspection station. The product is a capacitor used in most common household and commercial AC compressors. The objective of the system will be to detect when the product has arrived, take a picture of it

with an imaging device, and determine if the orientation of the capacitor is correct (reference Figure 1-1). If the system determines that the orientation is not correct, the robotic manipulator will pick the capacitor up and rotate it until it reaches the correct orientation. Then the manipulator will place the capacitor back in the boat and repeat the process until canceled.

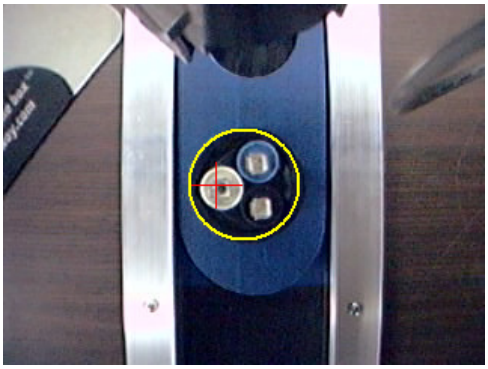


Figure 1-1 Capacitor Image

2 ROBOT DESIGN

2.1 Introduction

The most common manipulators (mechanical components and configuration of the robot) are sold in five arm kinematic arrangements and may be classified into four major categories:

- Cartesian – Three prismatic joints.
- Cylindrical – One revolute and two prismatic joints.
- Spherical – Two revolute and one prismatic joint.
- Articulated – Three revolute joints.

In an n-degree of freedom robotic manipulator, typically the first three joints will operate as the arm and the remaining joints operate as the wrist. It has become convention in the development of the direct kinematic control equations that the arm kinematics and the wrist kinematics be developed separately. The use of superposition gives the total direct kinematics of the manipulator.

The choice of the arm manipulator configuration should be the first step in the design process. A manipulator

configuration selection can be easy or very complicated. For the particular process chosen to automate, a cylindrical manipulator configuration has been chosen. The reason it is called a cylindrical manipulator is because the configuration of the links and joints envelope the motion to be in the shape of a cylinder (reference Figure 2-1).

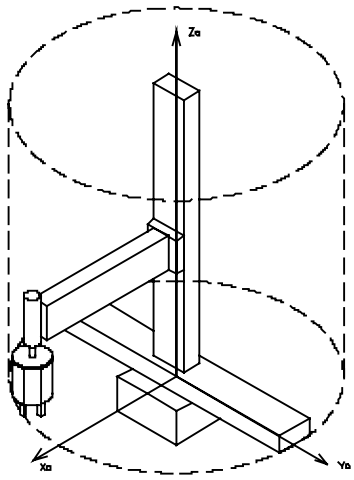


Figure 2-1 Manipulator Envelope

This envelope will be a perfect working space for the automation of the process used in this thesis topic. Some other configurations may work, but are not ideal for the process being automated. Some of the reasons being; the cartesian manipulator configuration is limited to a single plane, and will limit the growth capability of the system, and the spherical and fully articulated are more complicated and cost more.

This particular arm configuration consists of one revolute joint and two prismatic joints, as the cylindrical manipulator definition implies. The successive motions this manipulator will make are revolute, linear in the horizontal direction, and linear in the vertical direction. The wrist configuration consists of one revolute joint. Therefore the total degree of freedom of the system is four.

The next step after a configuration has been selected is to develop the kinematic equations for the arm and then develop the kinematic equations for the wrist. Kinematics is the branch of mechanics that deals with the motion of rigid bodies without reference to their masses or forces [2]. Putting the two solutions together using the superposition principle will produce a complete kinematic solution of the system. Thus enabling one to specify the location and the orientation of the end-effector with respect to the base coordinate frame. Kinematics also play a very important role in the development of the dynamics.

The next step after developing the kinematics would be to determine the generalized joint velocities, and their relationship to the linear velocities, and angular velocities. This is accomplished by developing the Jacobian matrix. The Jacobian is a matrix-valued function and can be thought of as

the vector version of the ordinary derivative of a scalar function [5].

The next and final step for the manipulator design is to develop the dynamic equations of the system. The dynamic equations describe the motion of the system by means of a mathematical model. The equations describe the evolution of the motion with respect to time. This is a very important part of the design and development of any time changing varying system. From the dynamic equations, one could model the system as closely and accurately to the actual system as one would like to get, so long as all the dynamics have a mathematical solution that exists.

These steps are the basic ingredients of a manipulator system design. Through the completion of each step, a suitable control system can be selected and simulated. Each step is described through the next few sections.

2.2 Kinematics

A systematic solution has been developed to derive the cartesian frames attached to each link (joint) of a manipulator. The commonly used convention is called the Denavit-Hartenberg (D-H) convention. This method developed in 1955, has been widely accepted as the procedure of choice for the generation of

the kinematic relations between joints and ultimately relates the end-effector to the base frame (world coordinate system). The procedure applies well to systems having greater than 2 degree-of-freedom (DOF) and less then 6 DOF.

2.2.1 Denavit-Hartenberg Representation

In the D-H convention, each homogeneous transformation matrix is called A_i, and is represented as a product of four primary transformations [5]:

$$A^{i}_{i-1} = Rot_{x,\alpha_i} * Trans_{x,a_i} * Rot_{z,\theta_i} * Trans_{z,d_i} = T_{\alpha_i} T_{a_i} T_{\theta_i} T_{d_i} \tag{2.1}$$

$$A^{i}_{i-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C\alpha_i & -S\alpha_i & 0 \\ 0 & S\alpha_i & C\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C\theta_i & -S\theta_i & 0 & 0 \\ S\theta_i & C\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A^{i}_{i-1} = \begin{bmatrix} C\theta_i & -S\theta_i & 0 & a_i \\ C\alpha_i S\theta_i & C\theta_i C\alpha_i & -S\alpha_i & -d_i S\alpha_i \\ S\theta_i S\alpha_i & S\alpha_i C\theta_i & C\alpha_i & d_i C\alpha_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The four parameters $\theta_i, \alpha_i, a_i, d_i$ are parameters of link *i* and joint *i*.

- a_i = Distance between z_i axis and z_{i-1} axis along x_i.
- d_i = Distance between x_i axis and x_{i-1} axis along z_i. This parameter is a variable for a prismatic joint.
- α_i = The angle between the positive z_{i-1} to the positive z_i axis, about the positive x_i axis.

- θ_i = The angle between the positive x_{i-1} and positive x_i axis, about the positive z_{i-1} axis. This parameter is a variable for a revolute joint.

The mapping of frame *i* coordinates to frame *i-1* is obtained by making a rotation of α degrees about x_i, followed by a translation of a units along x_i, followed by a rotation of θ degrees about z_{i-1}, and finally a translation of d units along z_{i-1}. To help in the development of the transformation matrices, a D-H table is constructed with the links and joint/link parameters.

2.2.2 Cylindrical Manipulator Kinematics

The cylindrical manipulator that is used for this automation system has the coordinate frames defined as shown in Figure 2-2. The frames are assigned to each link and joint by using the D-H method. The assignment of a coordinate frame to a link can be chosen systematically. Many books that explain the D-H theory and method explain a systematic approach to the determination of each link's coordinate frame. Many have a different approach, but the final transformation from end-effector to base frame should always be the same.

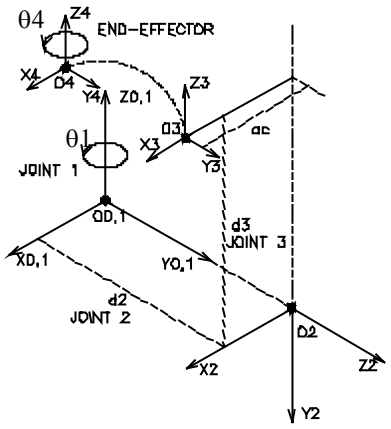


Figure 2-2 Coordinate Frames

As long as the right hand rule is followed and consistency between each coordinate frame is carried out, the homogeneous transformation from end-effector to base frame will be correct.

For a cylindrical manipulator, the arm joint variables are defined as, θ_1 for the revolute joint, d_2 for the horizontal prismatic joint and d_3 for the vertical prismatic joint. The wrist, which is attached to frame three has only one revolute joint defined by the angle of rotation about z_4. The z_4 axis is the symmetrical axis of the gripper. The variable to define this rotation is θ_4.

Using the D-H method, the development of the cylindrical arm kinematics will be carried out first and then followed by the development of the wrist kinematics. Table 2-1 list each link frame characterized by the kinematic parameters.

Table 2-1 D-H Parameter Table

Links	θ_i	α_i	a_i	d_i
1	θ_1	0	0	0
2	0	-90	0	d_2
3	0	90	a_3	d_3
Gripper	θ_4	0	0	0

Using the D-H homogeneous transformation matrix in Equation (2.2) and the D-H parameter table, each frame *i* can be translated back to frame *i-1*.

$$A^{i}_{i-1} = \begin{bmatrix} C\theta_i & -S\theta_i & 0 & a_i \\ C\alpha_i S\theta_i & C\theta_i C\alpha_i & -S\alpha_i & -d_i S\alpha_i \\ S\theta_i S\alpha_i & S\alpha_i C\theta_i & C\alpha_i & d_i C\alpha_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{Note: } \begin{aligned} c_1 &= \cos(\theta_1) \\ s_1 &= \sin(\theta_1) \\ c90 &= \cos(90) \\ s90 &= \sin(90) etc... \end{aligned} \tag{2.2}$$

Frame 1 to 0:

$$A^{1}_{0} = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 c0 & c_1 c0 & -s0 & 0 \\ s0 s_1 & c_1 s0 & c0 & c0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

Frame 2 to 1:

$$A^{2}_{1} = \begin{bmatrix} c0 & -s0 & 0 & 0 \\ s0c90 & c0c-90 & -s90 & -s90d_2 \\ s90s0 & c0s-90 & c-90 & c-90d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & d_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Frame 3 to 2

$$A^3{}_2 = \begin{bmatrix} c0 & -s0 & 0 & a_3 \\ s0c90 & c0c90 & -s90 & -s90d_3 \\ s90s0 & c0s90 & c90 & c90d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & ac \\ 0 & 0 & -1 & -d_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.5)$$

From Equations 2.3 - 2.5, the transformation matrix of each link to the base frame are equated to be:

$$T^1{}_0 = A^1{}_0 \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T^2{}_0 = (A^1{}_0)(A^2{}_1) = \begin{bmatrix} c_1 & 0 & -s_1 & -s_1 d_2 \\ s_1 & 0 & c_1 & c_1 d_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_0^3 = A_0^1 A_1^2 A_2^3 = \begin{bmatrix} c_1 & -s_1 & 0 & c_1 a_3 - s_1 d_2 \\ s_1 & c_1 & 0 & s_1 a_3 + c_1 d_2 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The wrist assembly consists of one revolute joint and a gripper. The transformation matrix to link three is:

$$A_3^4 = \begin{bmatrix} c_4 & -s_4 & 0 & 0 \\ s_4 c0 & c_4 c0 & -s0 & 0 \\ s0 s_4 & c_4 s0 & c0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_4 & -s_4 & 0 & 0 \\ s_4 & c_4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.6)$$

The total kinematic transformation for translation and orientation from the gripper axis to the base frame becomes:

$$T_0^4 = A_0^1 A_1^2 A_2^3 A_3^4 = \begin{bmatrix} c_1c_4 - s_1s_4 & -c_1s_4 - s_1c_4 & 0 & c_1a_3 - s_1d_2 \\ s_1c_4 + c_1s_4 & c_1c_4 - s_1s_4 & 0 & s_1a_3 + c_1d_2 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_0^4 = \begin{bmatrix} c_1c_4 - s_1s_4 & -c_1s_4 - s_1c_4 & 0 \\ s_1c_4 + c_1s_4 & c_1c_4 - s_1s_4 & 0 \\ 0 & 0 & 1 \end{bmatrix} orientation \qquad (2.7)$$

$$D_0^4 = \begin{bmatrix} c_1a_3 - s_1d_2 \\ s_1a_3 + c_1d_2 \\ d_3 \\ 1 \end{bmatrix} translation \quad T_0^4 = \begin{bmatrix} R_0^4 & D_0^4 \\ 0 & 1 \end{bmatrix} total\ transformation$$

With the transformation matrices computed, the location of any point on the cylindrical manipulator can be found relative to the base frame when the values of the joint variables θ_1, θ_4, d_2, and d_3 are known.

2.3 Velocity Relationships

The relationship of joint velocities to the linear and angular velocities with respect to the base frame are obtained through the mapping function of the Jacobian matrix. This is the next critical step in creating a complete model of a robotic control system.

The joint variables for the cylindrical 3-link manipulator arm are represented by q_i. For simplicity, the end-effector's revolute joint is neglected since frame four of the end-effector is parallel to frame three.

$$q = \begin{bmatrix} \theta_1 \\ d_2 \\ d_3 \end{bmatrix} \tag{2.8}$$

The position command variable is $X_i = \begin{bmatrix} x & y & z \end{bmatrix}^T$, which correlates with the linear translation of the end-effector D_0^4. The position command variables and coordinate command variables are related by $X_i = D(q)$. Differentiating with respect to time will give us the joint velocities related to the linear velocity:

$$\frac{dX_i}{dt} = \frac{dD(q)}{dq}\frac{dq}{dt} \ (chain\ rule)$$
$$J_v(q) = \frac{dD(q)}{dq} = \frac{\partial D(q)}{\partial q_i} \tag{2.9}$$
$$V_i = \dot{X}_i = J_v(q)\dot{q}$$

V_i is the linear velocity of the *ith* frame. To define the angular velocity of the coordinate frames let

$$w_0^i = \sum_{k=0}^{i} (R_{03}^k)\dot{q}_i \quad R_{03}^k \ represents\ 3rd\ column\ of\ R_0^k$$
$$\sum_{k=0}^{i} (R_{03}^k) \Rightarrow Jw_i = \begin{bmatrix} R_{03}^1 & R_{03}^2 \cdots\cdots R_{03}^i & 0 \end{bmatrix}$$

Therefore:

$$w_0^i = Jw_i \dot{q}_i \tag{2.10}$$

w_0^i represents the angular velocity of frame *i* with respect to the base frame.

Each link's frame defined for the manipulator system must be translated to the center of gravity (mass) of that link. The new coordinate transformation matrix from the frame defined at the center of gravity to the base frame will be used to develop the Jacobian matrices of each link. The center of gravity velocity is important for correct development of the dynamic equations of motion.

2.3.1 Link 1

Let $lx_1, ly_1, and\ lz_1$ represent the variables to the center of gravity (CG) from frame 1 of link 1. The CG transformation matrix will be represented as:

$$Tcg_1 = \begin{bmatrix} 1 & 0 & 0 & lx_1 \\ 0 & 1 & 0 & ly_1 \\ 0 & 0 & 1 & lz_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

The translation of the CG to the base frame is:

$$T^{1'}{}_{0} = T^{1}{}_{0} \times Tcg_1 = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & lx_1 \\ 0 & 1 & 0 & ly_1 \\ 0 & 0 & 1 & lz_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_1 & -s_1 & 0 & c_1 lx_1 - s_1 ly_1 \\ s_1 & c_1 & 0 & s_1 lx_1 + c_1 ly_1 \\ 0 & 0 & 1 & lz_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Jacobian for the translation velocity of frame one is derived as follows:

$$x = D_1(\theta_1, d_2, d_3) = c_1 lx_1 - s_1 ly_1$$
$$y = D_2(\theta_1, d_2, d_3) = s_1 lx_1 + c_1 ly_1$$
$$z = D_3(\theta_1, d_2, d_3) = lz_1$$
$$\dot{x} = \frac{\delta D_1}{\delta \theta_1} + \frac{\delta D_1}{\delta d_2} + \frac{\delta D_1}{\delta d_3} = (-s_1 lx_1 - c_1 ly_1) + 0 + 0$$
$$\dot{y} = \frac{\delta D_2}{\delta \theta_1} + \frac{\delta D_2}{\delta d_2} + \frac{\delta D_2}{\delta d_3} = (s_1 lx_1 - s_1 ly_1) + 0 + 0$$
$$\dot{z} = \frac{\delta D_3}{\delta \theta_1} + \frac{\delta D_3}{\delta d_2} + \frac{\delta D_3}{\delta d_3} = 0 + 0 + 0$$

Therefore the translation velocity Jacobian for frame one is:

$$Jv_1(q) = \begin{bmatrix} -s_1 lx_1 - c_1 ly_1 & 0 & 0 \\ c_1 lx_1 - s_1 ly_1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The translation velocity for frame one is:

$$V_1 = Jv_1(q)\dot{q} \tag{2.12}$$

The angular Jacobian velocity for frame one is:

$$Jw_1 = R^1_{03}$$
$$R^1_0 = \begin{bmatrix} c_1 & -s_1 & 0 \\ s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow Jw_1 = R^1_{03} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The angular velocity for frame one is:

$$w_1 = R_{03}^1 \dot{\theta} \tag{2.13}$$

The total Jacobian of frame one is represented as:

$$J_1(q) = \begin{bmatrix} J_{v1} \\ J_{w1} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} -s_1 lx_1 - c_1 ly_1 & 0 & 0 \\ c_1 lx_1 - s_1 ly_1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \end{bmatrix}$$

2.3.2 Link 2

Let $lx_2, ly_2, and\ lz_2$ represent the variables to the CG from frame two of link two. The CG transformation matrix will be represented as:

$$Tcg_2 = \begin{bmatrix} 1 & 0 & 0 & lx_2 \\ 0 & 1 & 0 & ly_2 \\ 0 & 0 & 1 & lz_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The translation of the CG of link 2 to the base frame is:

$$T^{2'}{}_0 = T^2{}_0 \times Tcg_2 = \begin{bmatrix} c_1 & 0 & -s_1 & -s_1 d_2 \\ s_1 & 0 & c_1 & c_1 d_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & lx_2 \\ 0 & 1 & 0 & ly_2 \\ 0 & 0 & 1 & lz_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_1 & 0 & -s_1 & c_1 lx_2 - s_1 lz_2 - s_1 d_2 \\ s_1 & 0 & c_1 & s_1 lx_2 + c_1 lz_2 + c_1 d_2 \\ 0 & -1 & 0 & ly_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Jacobian for the translation velocity of frame two is derived as follows:

$$x = D_1(\theta_1, d_2, d_3) = c_1 lx_2 - s_1 lz_2 - s_1 d_2$$
$$y = D_2(\theta_1, d_2, d_3) = s_1 lx_2 + c_1 lz_2 + c_1 d_2$$
$$z = D_3(\theta_1, d_2, d_3) = ly_2$$
$$\dot{x} = \frac{\delta D_1}{\delta \theta_1} + \frac{\delta D_1}{\delta d_2} + \frac{\delta D_1}{\delta d_3} = (-s_1 lx_2 - c_1 lz_2 - c_1 d_2) - s_1 + 0$$
$$\dot{y} = \frac{\delta D_2}{\delta \theta_1} + \frac{\delta D_2}{\delta d_2} + \frac{\delta D_2}{\delta d_3} = (c_1 lx_2 - s_1 lz_1 - s_1 d_2) + c_1 + 0$$
$$\dot{z} = \frac{\delta D_3}{\delta \theta_1} + \frac{\delta D_3}{\delta d_2} + \frac{\delta D_3}{\delta d_3} = 0 + 0 + 0$$

Therefore the Jacobian for link two is:

$$Jv_2(q) = \begin{bmatrix} -s_1 lx_2 - c_1 lz_2 - c_1 d_2 & -s_1 & 0 \\ c_1 lx_2 - s_1 lz_1 - s_1 d_2 & c_1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The translation velocity for frame two is:

$$V_2 = Jv_2(q)\dot{q} \tag{2.14}$$

The angular Jacobian velocity for frame two is:

$$Jw_2 = \begin{bmatrix} R_{03}^1 & R_{03}^2 \end{bmatrix}$$
$$R_0^2 = \begin{bmatrix} c_1 & 0 & -s_1 \\ s_1 & 0 & c_1 \\ 0 & -1 & 0 \end{bmatrix} \Rightarrow Jw_2 = \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

The angular velocity for frame two is:

$$w_2 = \begin{bmatrix} R_{03}^1 & R_{03}^2 \end{bmatrix}\dot{\theta} \tag{2.15}$$

The total Jacobian of frame two is represented as:

$$J_2(q) = \begin{bmatrix} J_{v2} \\ J_{w2} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} -s_1 lx_2 - c_1 lz_2 - c_1 d_2 & -s_1 & 0 \\ c_1 lx_2 - s_1 lz_1 - s_1 d_2 & c_1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \hline \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \end{bmatrix}$$

2.3.3 Link 3

Let $lx_3, ly_3, and\ lz_3$ represent the variables to the CG of link three. The CG transformation matrix is

$$Tcg_3 = \begin{bmatrix} 1 & 0 & 0 & lx_3 \\ 0 & 1 & 0 & ly_3 \\ 0 & 0 & 1 & lz_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The translation of the CG of link three to the base frame is:

$$T^{3'}{}_0 = T^3{}_0 \times Tcg_3 = \begin{bmatrix} c_1 & -s_1 & 0 & c_1 a_3 - s_1 d_2 \\ s_1 & c_1 & 0 & s_1 a_3 + c_1 d_2 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & lx_3 \\ 0 & 1 & 0 & ly_3 \\ 0 & 0 & 1 & lz_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T^{3'}{}_0 = \begin{bmatrix} c_1 & -s_1 & 0 & c_1 lx_3 - s_1 ly_3 + c_1 a_3 - s_1 d_2 \\ s_1 & c_1 & 0 & s_1 lx_3 + c_1 ly_3 + s_1 a_3 + c_1 d_2 \\ 0 & 0 & 1 & lz_3 + d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Jacobian for the translation velocity of frame three is derived as follows:

$$x = D_1(\theta_1, d_2, d_3) = c_1 lx_3 - s_1 ly_3 + c_1 a_3 - s_1 d_2$$
$$y = D_2(\theta_1, d_2, d_3) = s_1 lx_3 + c_1 ly_3 + s_1 a_3 + c_1 d_2$$
$$z = D_3(\theta_1, d_2, d_3) = lz_3 + d_3$$

$$\dot{x} = \frac{\delta D_1}{\delta \theta_1} + \frac{\delta D_1}{\delta d_2} + \frac{\delta D_1}{\delta d_3} = (-s_1 lx_3 - c_1 ly_3 - s_1 a_3 - c_1 d_2) - s_1 + 0$$
$$\dot{y} = \frac{\delta D_2}{\delta \theta_1} + \frac{\delta D_2}{\delta d_2} + \frac{\delta D_2}{\delta d_3} = (c_1 lx_3 - s_1 ly_3 + c_1 a_3 - s_1 d_2) + c_1 + 0$$
$$\dot{z} = \frac{\delta D_3}{\delta \theta_1} + \frac{\delta D_3}{\delta d_2} + \frac{\delta D_3}{\delta d_3} = 0 + 0 + 1$$

Therefore the Jacobian for link three is:

$$Jv_3(q) = \begin{bmatrix} -s_1 lx_3 - c_1 ly_3 - s_1 a_3 - c_1 d_2 & -s_1 & 0 \\ c_1 lx_3 - s_1 ly_3 + c_1 a_3 - s_1 d_2 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The translation velocity of frame three is:

$$V_3 = Jv_3(q)\dot{q} \tag{2.16}$$

The angular Jacobian velocity of frame three is:

$$Jw_3 = \begin{bmatrix} R_{03}^1 & R_{03}^2 & R_{03}^3 \end{bmatrix}$$
$$R_0^3 = \begin{bmatrix} c_1 & -s_1 & 0 \\ s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow Jw_3 = \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The angular velocity of frame three is:

$$w_3 = \begin{bmatrix} R_{03}^1 & R_{03}^2 & R_{03}^3 \end{bmatrix}\dot{\theta} \tag{2.17}$$

The total Jacobian of frame three is represented as:

$$J_3(q) = \begin{bmatrix} J_{v3} \\ J_{w3} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} -s_1 lx_3 - c_1 ly_3 - s_1 a_3 - c_1 d_2 & -s_1 & 0 \\ c_1 lx_3 - s_1 ly_3 + c_1 a_3 - s_1 d_2 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \hline \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \end{bmatrix}$$

The final Jacobian matrix below relates the end-effector translation and angular velocity to the joint velocities.

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \\ \hline \dot{W}_1 \\ \dot{W}_2 \\ \dot{W}_3 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} -s_1 lx_3 - c_1 ly_3 - s_1 a_3 - c_1 d_2 & -s_1 & 0 \\ c_1 lx_3 - s_1 ly_3 + c_1 a_3 - s_1 d_2 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \hline \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \end{bmatrix} \times \begin{bmatrix} \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3 \end{bmatrix} \qquad (2.18)$$

2.4 Dynamics

The dynamics of a robotic manipulator describe and allow one to model the time varying parameters of the system. Two methods have been developed to assist the creation of the dynamic equations. These two methods are the Newton-Euler formulation and the Euler-Lagrange equations. The latter is suited well for robotic dynamic equation development because the Lagrangian enables one to work with each axis individually. For

the Newton-Euler method, the forces exerted by one axis on to another must be taken into account. One can imagine how complicated this can get when a manipulator system consists of three or more links. A typical robotic configuration today, has an arrangement of six links or more. The following is the Lagrange formulation used to develop the dynamic equation for n-axis (link) manipulator.

$$L = K - P \tag{2.19}$$

The Lagrangian of the system is defined as the kinetic energy minus the potential energy as shown in equation (2.19). To be able to identify the Lagrangian of the complete manipulator, the kinetic and potential energies of each link must be determined. The sum of each individual Lagrangian will give the overall Lagrangian of the manipulator.

The kinetic energy of an object is the sum of two terms; the translation energy and the rotational energy of the object [5]. Both energies are calculated by concentrating the entire mass of the body at the center of gravity.

The translation part of the kinetic energy is defined as:

$$K_t = \frac{1}{2} v_c^T m v_c \tag{2.20}$$

Where v_c represents the velocity vector of a particle of mass m located at the CG.

The rotational part of the kinetic energy is defined as:

$$K_r = \frac{1}{2} w^T I w \tag{2.21}$$

Where *w* is the angular velocity vector about the CG and *I* is the inertia matrix evaluated around a coordinate frame whose origin is at the CG. Typically the moment of inertia I_i^i is calculated with respect to the coordinate frame attached to the object. Calculating the moment of inertia this way makes the calculations independent of the motion of the object. To get the moment of inertia with respect to the base frame, multiply I_i^i by R_0^i.

$$R_0^i I_i^i R_0^{iT} \tag{2.22}$$

The total kinetic energy becomes:

$$K = K_t + K_r \Rightarrow K = \frac{1}{2}(v_c^T m v_c + w^T I w) \tag{2.23}$$

The potential energy for a dynamic system made of rigid objects has gravity as the only source of potential energy [5]. The potential energy of an object with its mass located at its CG becomes:

$$P = m_i g z_i(q) \tag{2.24}$$

Where m_i is the mass of the link *i*, and $z_i(q)$ is the height of the CG of link *i*.

With the Lagrangian of the system defined, the next step is to develop the equations of motion for a robotic system with rigid links using the Euler-Lagrange equations of motion defined by:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right)-\frac{\partial L}{\partial q_i}=\tau_i \tag{2.25}$$

τ_i represents the generalized force (torque) vector at joint i.

Each link starting from the first link, will have its kinetic energy, potential energy, and Lagrangian derived individually. The summation of all the Lagrangians and equations of motion will give the overall general dynamic equations of motion defined by:

$$\begin{aligned} M(q)\ddot{q}+V(q,\dot{q})\dot{q}+G(q)&=\tau_i \\ N(q,\dot{q})&=V(q,\dot{q})\dot{q}+G(q) \\ M(q)\ddot{q}+N(q,\dot{q})&=\tau_i \end{aligned} \tag{2.26}$$

$M(q)$ is the inertia matrix multiplied by the second derivative of the generalized coordinates. $V(q,\dot{q})$ is the nonlinear matrix which contains centripetal and Coriolis terms. The centripetal terms involve the product of quadratic terms of the first derivatives of q, and the Coriolis terms involve the product of two generalized coordinates $q_i q_j$ where i does not

equal j. $G(q)$ is the matrix containing the differentiation of the potential energies of each link.

2.4.1 Link 1

A.) Kinetic Energy:

$$K_1 = \frac{1}{2} V^T{}_1 m_1 V_1 + \frac{1}{2} w^T{}_1 I_1 w_1$$

$$V_1 = Jv_1 \dot{q}$$

$$w_1 = Jw_1 \dot{q} \Rightarrow K_1 = \frac{1}{2} (Jv_1 \dot{q})^T m_1 Jv_1 \dot{q} + \frac{1}{2} (Jw_1 \dot{q})^T I_1 Jw_1 \dot{q}$$

$$(Jv_1 \dot{q})^T = \dot{q}^T Jv^T{}_1 \Rightarrow K_1 = \frac{1}{2} \dot{q}^T Jv^T{}_1 m_1 Jv_1 \dot{q} + \frac{1}{2} \dot{q}^T Jw^T{}_1 I_1 Jw_1 \dot{q}$$

$$K_1 = \frac{1}{2} \dot{q}^T (Jv^T{}_1 m_1 Jv_1 + Jw^T{}_1 I_1 Jw_1) \dot{q}$$

$$M_1(q) = Jv^T{}_1 m_1 Jv_1 + Jw^T{}_1 I_1 Jw_1 \Rightarrow K_1 = \frac{1}{2} \dot{q}^T M_1(q) \dot{q}$$

M1(q) is a symmetric positive definite inertia matrix that is manipulator configuration dependent. A positive definite matrix satisfies $\dot{q}^T M(q) \dot{q} > 0$. Hence there will always be positive kinetic energy in the manipulator system. A determinate of a positive definite matrix is always positive as well. Thus a positive definite real matrix is always nonsingular. A nonsingular matrix will always have an inverse.

B.) Potential Energy:

$$P_1 = m_1 g z_1(q)$$
$$z_1(q) = lz_1 \Rightarrow P_1 = m_1 g lz_1 \tag{2.27}$$

C.) Lagrangian:

$$L_1 = K_1 - P_1$$

$$K_1 = \frac{1}{2}\dot{q}^T M_1(q)\dot{q} \quad \dot{q} = \begin{bmatrix} \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3 \end{bmatrix} \quad P_1 = m_1 g lz_1$$

$$L_1(q,\dot{q}) = \frac{1}{2}\dot{q}^T M_1(q)\dot{q} - m_1 g lz_1 \Rightarrow \frac{1}{2}\begin{bmatrix} \dot{\theta}_1 & \dot{d}_2 & \dot{d}_3 \end{bmatrix} M_1(q) \begin{bmatrix} \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3 \end{bmatrix} - m_1 g lz_1$$

The Lagrangian can be rewritten as:

$$L_1(q,\dot{q}) = \sum_{k=1}^{3}\sum_{j=1}^{3} \dot{q}^T{}_{1k} M^1{}_{jk}(q)\dot{q}_{j1} - m_1 g z_1(q) \tag{2.28}$$

Where k represents the columns and j represents the rows. $M^1{}_{jk}$ is the SPD matrix for link one defined as.

$$M_1(q) = Jv^T{}_1 m_1 Jv_1 + Jw^T{}_1 I_1 Jw_1$$

$$Jv_1(q) = \begin{bmatrix} -s_1 lx_1 - c_1 ly_1 & 0 & 0 \\ c_1 lx_1 - s_1 ly_1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Jw_1(q) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$I_1 = R_0^1 I_1^1 R_0^{1T} \Rightarrow R_0^1 = \begin{bmatrix} c_1 & -s_1 & 0 \\ s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow I_1^1 = \begin{bmatrix} Ixx_1 & 0 & 0 \\ 0 & Iyy_1 & 0 \\ 0 & 0 & Izz_1 \end{bmatrix} \quad Ixx_1 = Iyy_1 = 0$$

$$I_1 = \begin{bmatrix} c_1 & -s_1 & 0 \\ s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Izz_1 \end{bmatrix} \begin{bmatrix} c_1 & s_1 & 0 \\ -s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Izz_1 \end{bmatrix}$$

To calculate I_1, link one is modeled after a cylinder and a rectangular cube as illustrated in Figure 2-3.

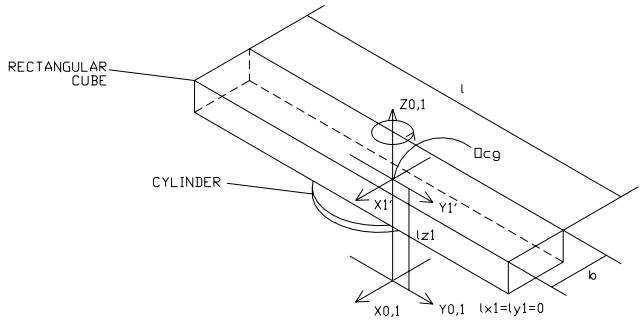


Figure 2-3 Link 1

Since the angular velocity occurs around the principal z-axis the estimated moment of inertia about z is the sum of the moment of inertia of a solid cylinder and a rectangular cube.

$$Iz_{cylinder} = \frac{1}{2} m_r r^2 \quad Iz_{rect} = \frac{m_l}{12}(l^2 + b^2)$$
$$Izz_1 = \frac{1}{2} m_r r^2 + \frac{m_l}{12}(l^2 + b^2) \tag{2.29}$$

Since frame 1 is defined so that the center of gravity is offset only by l_{z1} ($lx1=ly1=0$). Therefore the translation Jacobian is equal to zero. The inertia matrix $M_1(q)$ is equal to:

$$M_1(q) = Jv^T{}_1 m_1 Jv_1 + Jw^T{}_1 I_1 Jw_1 \Rightarrow Jv_1 = 0$$
$$M_1(q) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Izz_1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} Izz_1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.30}$$

The Lagrangian for link one becomes:

$$L_1(q,\dot{q}) = \frac{1}{2}\begin{bmatrix}\dot{\theta}_1 & \dot{d}_2 & \dot{d}_3\end{bmatrix}\begin{bmatrix}Izz_1 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}\begin{bmatrix}\dot{\theta}_1\\ \dot{d}_2\\ \dot{d}_3\end{bmatrix} - m_1 g l z_1 \qquad (2.31)$$

2.4.2 Link 2

A.) Kinetic Energy

$$K_2 = \frac{1}{2}V^T{}_2 m_2 V_2 + \frac{1}{2}w^T{}_2 I_2 w_2$$

$$V_2 = Jv_2\,\dot{q}$$

$$w_2 = Jw_2\,\dot{q} \Rightarrow K_2 = \frac{1}{2}(Jv_2\,\dot{q})^T m_2 Jv_2\,\dot{q} + \frac{1}{2}(Jw_2\,\dot{q})^T I_2 Jw_2\,\dot{q}$$

$$(Jv_2\,\dot{q})^T = \dot{q}^T\,Jv^T{}_2 \Rightarrow K_2 = \frac{1}{2}\dot{q}^T\,Jv^T{}_2 m_2 Jv_2\,\dot{q} + \frac{1}{2}\dot{q}^T\,Jw^T{}_2 I_2 Jw_2\,\dot{q}$$

$$K_2 = \frac{1}{2}\dot{q}^T(Jv^T{}_2 m_2 Jv_2 + Jw^T{}_2 I_2 Jw_2)\,\dot{q}$$

$$M_2(q) = Jv^T{}_2 m_2 Jv_2 + Jw^T{}_2 I_2 Jw_2 \Rightarrow K_2 = \frac{1}{2}\dot{q}^T\,M_2(q)\,\dot{q}$$

B.) Potential Energy

$$P_2 = m_2 g z_2(q) \qquad (2.32)$$

$$z_2(q) = ly_2 \Rightarrow P_2 = m_2 g l y_2$$

C.) Lagrangian

$$L_2(q,\dot{q}) = \frac{1}{2}\dot{q}^T M_2(q)\dot{q} - m_2 g l y_2$$

$$L_2(q,\dot{q}) = \frac{1}{2}\begin{bmatrix}\dot{\theta} & \dot{d}_2 & \dot{d}_3\end{bmatrix} M_2(q) \begin{bmatrix}\dot{\theta}\\ \dot{d}_2\\ \dot{d}_3\end{bmatrix} - m_2 g l y_2$$

Rewriting the Lagrangian as:

$$L_2(q,\dot{q}) = \sum_{k=1}^{3}\sum_{j=1}^{3} \dot{q}^T{}_{1k} M^2{}_{jk}(q)\dot{q}_{j1} - m_2 g l y_2$$

$M^2{}_{jk}$ is the SPD matrix for link 2. The inertia matrix is calculated to be:

$$M_2(q) = Jv^T{}_2 m_2 Jv_2 + Jw^T{}_2 I_2 Jw_2$$

$$Jv_2(q) = \begin{bmatrix} -s_1 lx_2 - c_1 lz_2 - c_1 d_2 & -s_1 & 0 \\ c_1 lx_2 - s_1 lz_1 - s_1 d_2 & c_1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Jw_2 = \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$I_2 = R_0^2 I_2^2 R_0^{2T} \Rightarrow R_0^2 = \begin{bmatrix} c_1 & 0 & -s_1 \\ s_1 & 0 & c_1 \\ 0 & -1 & 0 \end{bmatrix} \Rightarrow I_2^2 = \begin{bmatrix} Ixx_2 & 0 & 0 \\ 0 & Iyy_2 & 0 \\ 0 & 0 & Izz_2 \end{bmatrix} \quad Ixx_2 = Izz_2 = 0$$

$$I_2 = \begin{bmatrix} c_1 & 0 & -s_1 \\ s_1 & 0 & c_1 \\ 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 0 & 0 \\ 0 & Iyy_2 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} c_1 & s_1 & 0 \\ 0 & 0 & -1 \\ -s_1 & c_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Iyy_2 \end{bmatrix}$$

To calculate I_2, link two is modeled after a rectangular cube as illustrated in Figure 2-4.

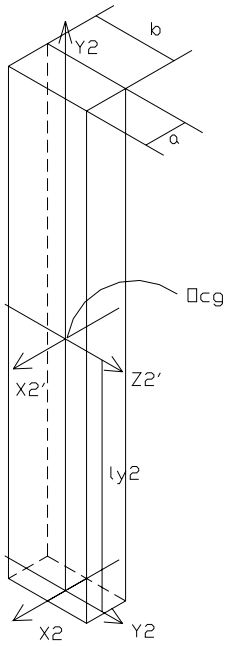


Figure 2-4 Link 2

Since the angular velocity occurs around the principal y-axis the estimated moment of inertia about *y* is:

$$Iyy_2 = \frac{m_2}{12}(a^2 + b^2) \tag{2.33}$$

Frame two is defined so that the center of gravity is offset only by l_{y2}, therefore *lx2=lz2=0*.

The inertia matrix $M_2(q)$ is equal to:

$$M_2(q) = Jv^T{}_2 m_2 Jv_2 + Jw^T{}_2 I_2 Jw_2$$

$$M_2(q) = \begin{bmatrix} -c_1 d_2 & -s_1 d_2 & 0 \\ -s_1 & c_1 & 0 \\ 0 & 0 & 0 \end{bmatrix} m_2 \begin{bmatrix} -c_1 d_2 & -s_1 & 0 \\ -s_1 d_2 & c_1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 \\ -s_1 & c_1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Iyy_2 \end{bmatrix} \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$M_2(q) = \begin{bmatrix} m_2 c_1^2 d_2^2 + m_2 s_1^2 d_2^2 + Iyy_2 & 0 & 0 \\ 0 & m_2(c_1^2 + s_1^2) & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} m_2 d_2^2 (c_1^2 + s_1^2) + Iyy_2 & 0 & 0 \\ 0 & m_2(c_1^2 + s_1^2) & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$M_2(q) = \begin{bmatrix} m_2 d_2^2 + Iyy_2 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The Lagrangian for link two becomes:

$$L_2(q, \dot{q}) = \frac{1}{2} \begin{bmatrix} \dot{\theta}_1 & \dot{d}_2 & \dot{d}_3 \end{bmatrix} \begin{bmatrix} m_2 d_2^2 + Iyy_2 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3 \end{bmatrix} - m_2 g l y_2$$

2.4.3 Link 3

A.) Kinetic Energy

$$K_3 = \frac{1}{2} V^T{}_3 m_3 V_3 + \frac{1}{2} w^T{}_3 I_3 w_3$$

$$V_3 = Jv_3 \dot{q}$$

$$w_3 = Jw_3 \dot{q} \Rightarrow K_3 = \frac{1}{2}(Jv_3 \dot{q})^T m_3 Jv_3 \dot{q} + \frac{1}{2}(Jw_3 \dot{q})^T I_3 Jw_3 \dot{q}$$

$$(Jv_3 \dot{q})^T = \dot{q}^T Jv^T{}_3 \Rightarrow K_3 = \frac{1}{2}\dot{q}^T Jv^T{}_3 m_3 Jv_3 \dot{q} + \frac{1}{2}\dot{q}^T Jw^T{}_3 I_3 Jw_3 \dot{q}$$

$$K_3 = \frac{1}{2}\dot{q}^T (Jv^T{}_3 m_3 Jv_3 + Jw^T{}_3 I_3 Jw_3)\dot{q}$$

$$M_3(q) = Jv^T{}_3 m_3 Jv_3 + Jw^T{}_3 I_3 Jw_3 \Rightarrow K_3 = \frac{1}{2}\dot{q}^T M_3(q)\dot{q}$$

B.) Potential Energy

$$P_3 = m_3 g z_3(q)$$
$$z_3(q) = d_3 \Rightarrow P_3 = m_3 g d_3 \qquad (2.34)$$

C.) Lagrangian

$$L_3(q,\dot{q}) = \frac{1}{2}\dot{q}^T M_3(q)\dot{q} - m_3 g d_3$$

$$L_3(q,\dot{q}) = \frac{1}{2}\begin{bmatrix}\dot{\theta} & \dot{d}_2 & \dot{d}_3\end{bmatrix} M_3(q) \begin{bmatrix}\dot{\theta} \\ \dot{d}_2 \\ \dot{d}_3\end{bmatrix} - m_3 g d_3$$

The Lagrangian can be rewritten as:

$$L_3(q,\dot{q}) = \sum_{k=1}^{3}\sum_{j=1}^{3} \dot{q}^T{}_{1k} M^3{}_{jk}(q)\dot{q}_{j1} - m_3 g z_3(q) \qquad (2.35)$$

$M^3{}_{jk}$ is the SPD matrix for link 3.

The inertia matrix is calculated to be:

$$M_3(q) = Jv^T{}_3 m_3 Jv_3 + Jw^T{}_3 I_3 Jw_3$$

$$Jv_3(q) = \begin{bmatrix} -s_1 lx_3 - c_1 ly_3 - s_1 a_3 - c_1 d_2 & -s_1 & 0 \\ c_1 lx_3 - s_1 ly_3 + c_1 a_3 - s_1 d_2 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad Jw_3 = \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$I_3 = R_0^3 I_3^3 R_0^{3T} \Rightarrow R_0^3 = \begin{bmatrix} c_1 & -s_1 & 0 \\ s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow I_3^3 = \begin{bmatrix} Ixx_3 & 0 & 0 \\ 0 & Iyy_3 & 0 \\ 0 & 0 & Izz_3 \end{bmatrix} \quad Ixx_3 = Iyy_3 = 0$$

$$I_3 = \begin{bmatrix} c_1 & -s_1 & 0 \\ s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Izz_3 \end{bmatrix} \begin{bmatrix} c_1 & s_1 & 0 \\ -s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Izz_3 \end{bmatrix}$$

To calculate I_3, link three is modeled after a rectangular cube as illustrated in Figure 2-5.

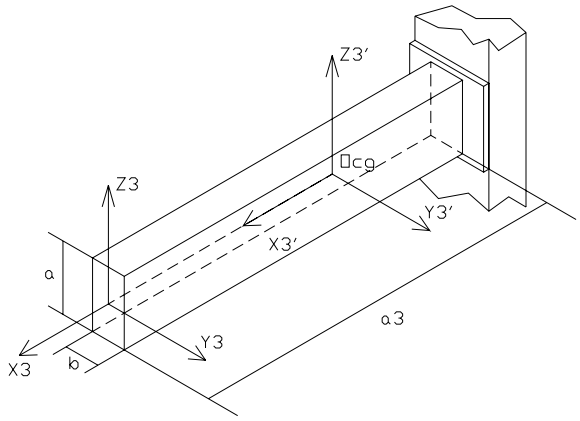


Figure 2-5 Link 3

Since the angular velocity occurs around the principal z-axis the estimated moment of inertia about z is:

$$Izz_3 = \frac{m_3}{12}\left((\frac{a_3}{2})^2 + b^2 \right) \qquad (2.36)$$

Frame three is defined so that the translation of the center of gravity from frame three is offset only by $l_{x3}=-a_3/2$, therefore $ly3=lz3=0$.

The inertia matrix $M_3(q)$ is equal to:

$$M_3(q) = Jv^T{}_3 m_3 Jv_3 + Jw^T{}_3 I_3 Jw_3$$

$$M_3(q) = \begin{bmatrix} s_1(\frac{a_3}{2}) - s_1 a_3 - c_1 d_2 & c_1(\frac{a_3}{2}) + c_1 a_3 - s_1 d_2 & 0 \\ -s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} m_3 \begin{bmatrix} s_1(\frac{a_3}{2}) - s_1 a_3 - c_1 d_2 & -s_1 & 0 \\ c_1(\frac{a_3}{2}) + c_1 a_3 - s_1 d_2 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} +$$

$$\begin{bmatrix} 0 & 0 & 1 \\ -s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Izz_3 \end{bmatrix} \begin{bmatrix} 0 & -s_1 & 0 \\ 0 & c_1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$M_3(q) = \begin{bmatrix} m_3\left(\left(\frac{-s_1 a_3}{2} - c_1 d_2\right)^2 + \left(\frac{c_1 a_3}{2} - s_1 d_2\right)^2\right) + Izz_3 & -m_3\left(\left(\frac{-s_1 a_3}{2} - c_1 d_2\right)s_1 - \left(\frac{c_1 a_3}{2} - s_1 d_2\right)c_1\right) & Izz_3 \\ -m_3\left(\left(\frac{-s_1 a_3}{2} - c_1 d_2\right)s_1 - \left(\frac{c_1 a_3}{2} - s_1 d_2\right)c_1\right) & m_3(c_1^2 + s_1^2) & 0 \\ Izz_3 & 0 & m3 + Izz_3 \end{bmatrix}$$

$$Identity: (c_1^2 + s_1^2) = 1 \qquad M_3(q) = \begin{bmatrix} m_3\left(\frac{a_3{}^2}{4} + d_2{}^2\right) + Izz_3 & \frac{m_3 a_3}{2} & Izz_3 \\ \frac{m_3 a_3}{2} & m_3 & 0 \\ Izz_3 & 0 & m_3 + Izz_3 \end{bmatrix}$$

The Lagrangian for link three becomes:

$$L_3(q,\dot{q}) = \frac{1}{2}\begin{bmatrix}\dot{\theta}_1 & \dot{d}_2 & \dot{d}_3\end{bmatrix}\begin{bmatrix} m_3\left(\frac{a_3^{\,2}}{4}+d_2^{\,2}\right)+Izz_3 & \frac{m_3 a_3}{2} & Izz_3 \\ \frac{m_3 a_3}{2} & m_3(c_1^2+s_1^2) & 0 \\ Izz_3 & 0 & m_3+Izz_3 \end{bmatrix}\begin{bmatrix}\dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3\end{bmatrix} - m_3 g d_3$$

The total Lagrangian of the complete manipulator assembly is summation of all three Lagrangians. The total Lagrangian becomes:

$$L(q,\dot{q}) = \frac{1}{2}[\dot{\theta}_1^2\left(I_{ZZ1}+m_2 d_2^2+I_{YY2}+m_3\left(\frac{a_3^2}{4}+d_2^2\right)+I_{ZZ3}\right)+ \qquad (2.37)$$

$$\dot{d}_2^2\left(m_2+m_3\right)+\dot{d}_3^2\left(m_3+I_{ZZ3}\right)+\dot{\theta}_1\dot{d}_2\, m_3 a_3+2\dot{\theta}_1\dot{d}_3\, I_{ZZ3}]-$$

$$g\left(m_1 l_{Z1}+m_2 l_{Y2}+m_3 d_3\right)$$

$$L(q,\dot{q}) = \sum_{i=1}^{3}\sum_{k=1}^{3}\sum_{j=1}^{3} q^{\dot{}}_{1k}^{T} M^{i}{}_{jk}(q)\,\dot{q}_{j1} - \sum_{i}^{3} m_i g z_i(q)$$

2.4.4 Equations of Motion

With the Lagrangian of the system defined, the next step is to develop the equations of motion for a robotic system with rigid links using the Euler-Lagrange equations of motion defined by:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \tau_i$$

Joint 1:

$$\tau_1 = \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_1}\right) - \frac{\partial L}{\partial \theta_1} \tag{2.38}$$

From equation (2.38) let:

$$y = \frac{\partial L}{\partial \dot{\theta}_1} = \left(I_{ZZ1} + m_2 d_2^2 + I_{YY2} + m_3\left(\frac{a_3^2}{4} + d_2^2\right) + I_{ZZ3}\right)\dot{\theta}_1 + \frac{m_3 a_3}{2}\dot{d}_2 + I_{ZZ3}\dot{d}_3$$

In generic terms:

$$y = \frac{\partial L}{\partial \dot{\theta}_1} = F\left(q, \dot{q}\right) \tag{2.39}$$

Applying the Chain Rule to equation (2.39):

$$\frac{dy}{dt} = \frac{\partial F}{\partial q}\frac{dq}{dt} + \frac{\partial F}{\partial \dot{q}}\frac{d\dot{q}}{dt} \tag{2.40}$$

$$\frac{\partial F}{\partial q}\frac{dq}{dt} = 2\left(m_2 + m_3\right) d_2 \dot{\theta}_1 \dot{d}_2$$

$$\frac{\partial F}{\partial \dot{q}}\frac{d\dot{q}}{dt} = \left(I_{ZZ1} + m_2 d_2^2 + I_{YY2} + m_3\left(\frac{a_3^2}{4} + d_2^2\right) + I_{ZZ3}\right)\ddot{\theta}_1 + \frac{m_3 a_3}{2}\ddot{d}_2 + I_{ZZ3}\ddot{d}_3$$

Also:

$$\frac{\partial L}{\partial \theta_1} = 0 \tag{2.41}$$

From equation (2.40) and (2.54) the torque for joint one is:

$$\tau_1 = \begin{bmatrix} \alpha & \frac{m_3 a_3}{2} & I_{ZZ3} \end{bmatrix}\begin{bmatrix} \ddot{\theta}_1 \\ \ddot{d}_2 \\ \ddot{d}_3 \end{bmatrix} + \begin{bmatrix} 0 & 2d_2\dot{\theta}_1(m_2 + m_3) & \frac{m_3 \dot{d}_2}{2} \end{bmatrix}\begin{bmatrix} \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3 \end{bmatrix} \tag{2.42}$$

Joint 2:

$$\tau_2 = \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{d}_2}\right) - \frac{\partial L}{\partial d_2} \tag{2.43}$$

From equation (2.43) let:

$$y = \frac{\partial L}{\partial \dot{d}_2} = m_3 a_3 \dot{\theta}_1 + 2\left(m_2 + m_3\right)\dot{d}_2$$

In generic terms:

$$y = \frac{\partial L}{\partial \dot{d}_2} = F\left(q, \dot{q}\right) \tag{2.44}$$

Applying the Chain Rule to equation (2.44):

$$\frac{dy}{dt} = \frac{\partial F}{\partial q}\frac{dq}{dt} + \frac{\partial F}{\partial \dot{q}}\frac{d\dot{q}}{dt} \tag{2.45}$$

$$\frac{\partial F}{\partial q}\frac{dq}{dt} = 0$$

$$\frac{\partial F}{\partial \dot{q}}\frac{d\dot{q}}{dt} = m_3 a_3 \ddot{\theta}_1 + 2\left(m_2 + m_3\right)\ddot{d}_2$$

Also:

$$\frac{\partial L}{\partial d_2} = \left(m_2 + m_3\right) d_2 \dot{\theta}_1^2 \tag{2.46}$$

From equation (2.45) and (2.46) the torque for joint two is:

$$\tau_2 = \begin{bmatrix} m_3 a_3 & 2\left(m_2 + m_3\right) & 0 \end{bmatrix}\begin{bmatrix} \ddot{\theta}_1 \\ \ddot{d}_2 \\ \ddot{d}_3 \end{bmatrix} + \begin{bmatrix} -d_2 \dot{\theta}_1 (m_2 + m_3) & 0 & 0 \end{bmatrix}\begin{bmatrix} \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3 \end{bmatrix} \tag{2.47}$$

Joint 3:

$$\tau_3 = \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{d}_3}\right) - \frac{\partial L}{\partial d_3} \tag{2.48}$$

From equation (2.48) let:

$$y = \frac{\partial L}{\partial \dot{d}_3} = 2I_{ZZ3}\dot{\theta}_1 + 2\left(m_3 + I_{ZZ3}\right)\dot{d}_3$$

In generic terms:

$$y = \frac{\partial L}{\partial \dot{d}_3} = F\left(q, \dot{q}\right) \tag{2.49}$$

Applying the Chain Rule to equation (2.62):

$$\frac{dy}{dt} = \frac{\partial F}{\partial q}\frac{dq}{dt} + \frac{\partial F}{\partial \dot{q}}\frac{d\dot{q}}{dt} \tag{2.50}$$

$$\frac{\partial F}{\partial q}\frac{dq}{dt} = 0$$

$$\frac{\partial F}{\partial \dot{q}}\frac{d\dot{q}}{dt} = 2I_{ZZ3}\ddot{\theta}_1 + 2\left(m_3 + I_{ZZ3}\right)\ddot{d}_3$$

Also:

$$\frac{\partial L}{\partial d_2} = -gm_3 \tag{2.51}$$

From equation (2.50) and (2.51) the torque for joint three is:

$$\tau_3 = \begin{bmatrix} 2I_{ZZ3} & 0 & 2\left(m_3 + I_{ZZ3}\right) \end{bmatrix} \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{d}_2 \\ \ddot{d}_3 \end{bmatrix} - gm_3 \tag{2.52}$$

The Euler-Lagrange equation of motion in general terms can be written as [4]:

$$M(q)\ddot{q} + N(q,\dot{q}) = \tau$$
$$N(q,\dot{q}) = V(q,\dot{q})\dot{q} + G(q) + F_d(\dot{q}) + F_s(\dot{q}) + T_d \qquad (2.53)$$

Where: T_d is the generalized input due to disturbances

$F_s(\dot{q})$ represents the unstructured friction effects.

$F_d(\dot{q})$ represents the viscous and/or dynamic friction.

$V(q,\dot{q})$ is the matrix containing centripetal and Coriolis terms.

For simplicity all outside forces are set equal to zero. Therefore:

$$N(q,\dot{q}) = V(q,\dot{q})\dot{q} + G(q)$$

The complete cylindrical manipulator equations of motion are represented in matrix form by Equation (2.54):

$$\begin{bmatrix} \alpha & \frac{m_3 a_3}{2} & I_{ZZ3} \\ m_3 a_3 & 2(m_2+m_3) & 0 \\ 2I_{ZZ3} & 0 & 2(m_3+I_{ZZ3}) \end{bmatrix} \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{d}_2 \\ \ddot{d}_3 \end{bmatrix} + \begin{bmatrix} 0 & 2d_2\dot{\theta}_1(m_2+m_3) & \frac{m_3\dot{d}_2}{2} \\ -d_2\dot{\theta}_1(m_2+m_3) & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ m_3 g \end{bmatrix} = \begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} \qquad (2.54)$$

Where:

$$\alpha = I_{ZZ1} + m_2 d_2^2 + I_{YY2} + m_3\left(\frac{a_3^2}{4} + d_2^2\right) + I_{ZZ3}$$

2.5 Control Strategy

Linear control laws such as computed torque control, proportional-derivative (PD), and proportional-integral-derivative (PID) are commonly used control strategies for controlling robotic manipulators. Robotic manipulators are highly nonlinear. Using linear control laws such as PD and PID control will generally show stability only in a finite region. To be able to apply the fully developed linear control theory, the nonlinear robotic system must be linearized. A common technique used to linearize a nonlinear system is feedback linearization. This technique will be applied to the cylindrical manipulator system discussed in this manuscript.

2.5.1 Feedback Linearization

Feedback linearization cancels all the nonlinear terms by doing a nonlinear transformation and nonlinear feedback control. When this is implemented successfully, all the linear control laws apply. A fundamental linear control for a feedback control system is the proportional-plus integral-plus-derivative PID controller. Most often it is not necessary to implement all three terms. To demonstrate a straightforward implementation of the PID controller, a proportional-plus-derivative (PD) control

will be designed and simulated for the cylindrical robotic system.

Consider the general form of the dynamic equations for the cylindrical rigid-body robot:

$$\tau = M(q)\ddot{q} + V(q,\dot{q})\dot{q} + G(q) \tag{2.55}$$

Where:

$$\begin{aligned} N(q,\dot{q}) &= V(q,\dot{q})\dot{q} + G(q) \\ \tau &= M(q)\ddot{q} + N(q,\dot{q}) \end{aligned} \tag{2.56}$$

$N(q,\dot{q})$ and $M(q)$ are the nonlinear terms that must be canceled in order to implement the PD linear control technique.

Solving the second order differential equation for $\ddot{q}$:

$$\ddot{q} = M^{-1}(q)(-N(q,\dot{q}) + \tau) \tag{2.57}$$

Let $\tau = N(q,\dot{q}) + M(q)u$ where u is the output of the PD compensator used to control the torque output. For a PD compensator (controller):

$$u = K_D\dot{e} + K_p e \tag{2.58}$$

Substitute u for τ and solve for $\ddot{q}$:

$$\begin{aligned} \tau &= N(q,\dot{q}) + M(q)\left(K_D\dot{e} + K_p e\right) \\ \ddot{q} &= M^{-1}(q)\left(-N(q,\dot{q}) + N(q,\dot{q}) + M(q)\left(K_D\dot{e} + K_p e\right)\right) \\ \ddot{q} &= K_D\dot{e} + K_p e \qquad \therefore \ddot{q} = u \end{aligned}$$

Let q_d be the desired trajectory of the robot.

$$\begin{aligned} \ddot{q} &= K_D \dot{e} + K_p e \\ \ddot{q} &= (\ddot{q}_d + K_D(\dot{q}_d - \dot{q}) + K_P(q_d - q)) \\ 0 &= (\ddot{q}_d - \ddot{q}) + K_D(\dot{q}_d - \dot{q}) + K_P(q_d - q) \\ 0 &= \ddot{e} + K_D \dot{e} + K_P e \end{aligned} \tag{2.59}$$

For a typical second order differential equation the characteristic equation for the compensator is:

$$s^2 + 2\zeta\omega_n s + \omega_n^2 \tag{2.60}$$

Where: ζ is the damping ratio

ω_n is the natural frequency

Taking the Laplace transform of Equation (2.59) and assuming zero for the initial conditions, Equation (2.59) becomes:

$$s^2 + K_D s + K_P \tag{2.61}$$

From Equation (2.60) K_D and K_P are set equal to:

$$K_D = 2\zeta\omega_n \quad K_P = \omega_n^2$$

Repeated real roots are necessary for a critically damped system which will provide zero overshot. A critically damped system is ideal for robotic applications. For a critically damped system, set the damping ratio $\zeta = 1$.

Solving for K_D and K_P we have

$$K_D = 2\omega_n \qquad K_P = \omega_n^2 \tag{2.62}$$

The time constant is equal to $\tau = \frac{1}{\zeta\omega_n}$. To calculate the initial gains, a settling time of four time constants is required.

Let $t_s = 0.5s$. From Equation (2.62):

$$t_s = \frac{4}{\zeta\omega_n} \quad \zeta = 1 \Rightarrow t_s = \frac{4}{\omega_n}$$

$$\sqrt{Kp} = wn$$

$$0.5 = \frac{4}{\sqrt{K_P}} \Rightarrow 0.25 = \frac{16}{K_P} \Rightarrow K_P = 64$$

$$K_D = 2\sqrt{K_P} \Rightarrow K_D = 2\sqrt{64} \Rightarrow K_D = 16$$

The PD control will force the 2nd order differential equation $\ddot{q}$ poles in the left half plane to provide a stable control system.

$$\ddot{q} = \ddot{q}_d + 16(\dot{q}_d - \dot{q}) + 64(q_d - q) \qquad (2.63)$$

2.5.2 Simulation

To simulate the linearized nonlinear system using PD control, a closed loop control system was constructed using Matlab's Simulink software program. Figure 2-6 illustrates the constructed block diagram.

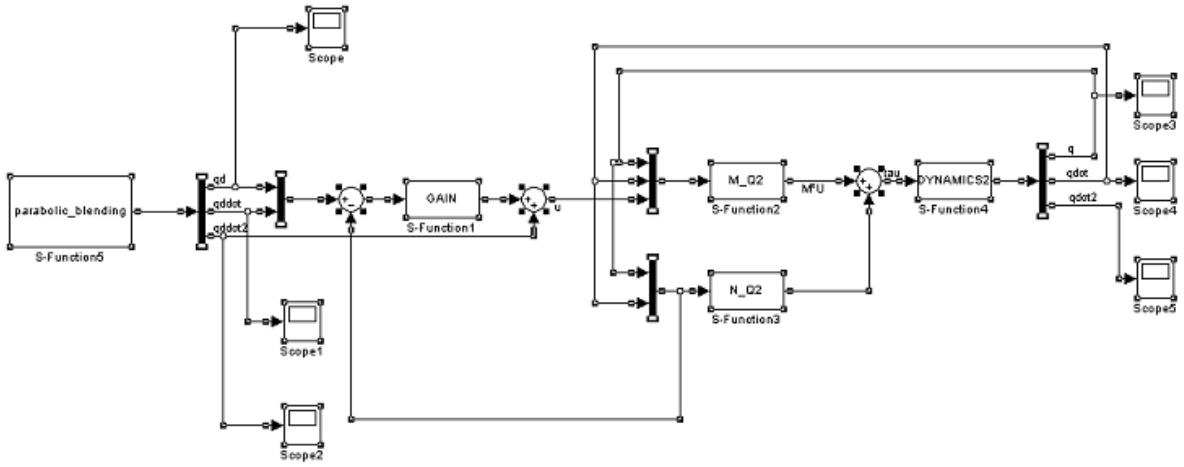


Figure 2-6 PD Simulation Block Diagram

The following constants have been developed for this model to demonstrate the control system stability. For link 1 reference Figure 2-7:

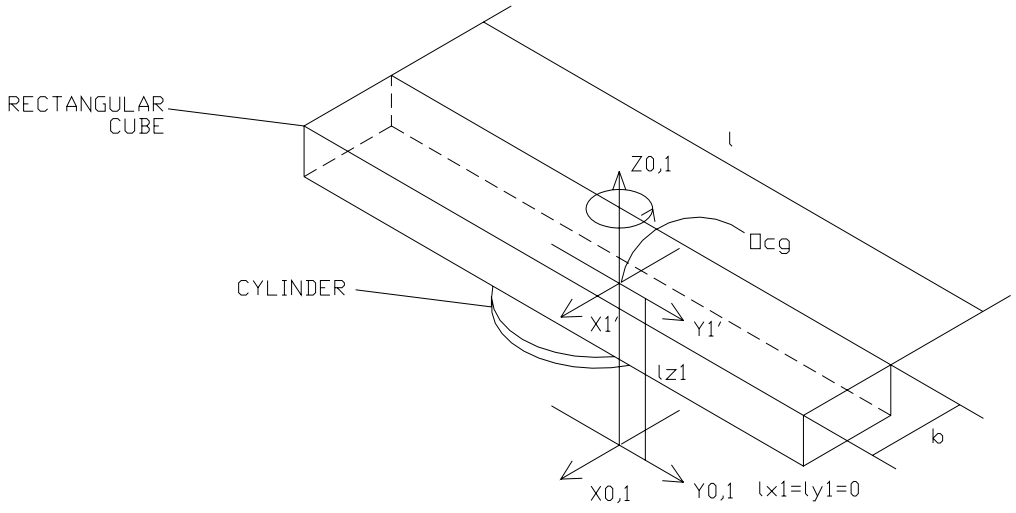


Figure 2-7 Link 1 Center of Gravity

The constructed parameters for link one are:

$$Izz_1 = \frac{1}{2} m_c r^2 + \frac{m_l}{12}(l^2 + b^2)$$
$$r = 0.0762\ m \quad l = 0.5080\ m$$
$$m_r = 0.5\ kg \quad b = 0.0889\ m$$
$$m_l = 4.8\ kg \Rightarrow m_1 = m_l + m_r = 5.3\ kg$$
$$Izz_1 = 0.0015\ kgm^2 + 0.1064\ kgm^2 = 0.1079\ kgm^2$$

For link 2 reference Figure 2-8:

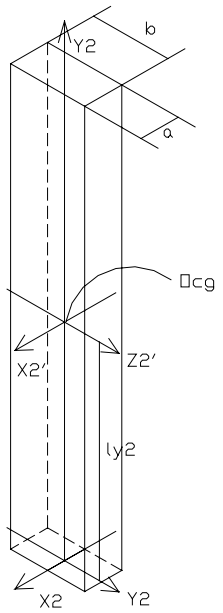


Figure 2-8 Link 2 Center of Gravity

The constructed parameters for link two are:

$$Iyy_2 = \frac{m_2}{12}(a^2 + b^2)$$
$$m_2 = 4.8\ kg$$
$$a = 0.0603\ m$$
$$b = 0.0889\ m$$
$$Iyy_2 = 0.0046\ kgm^2$$

For link 3 reference Figure 2-9:

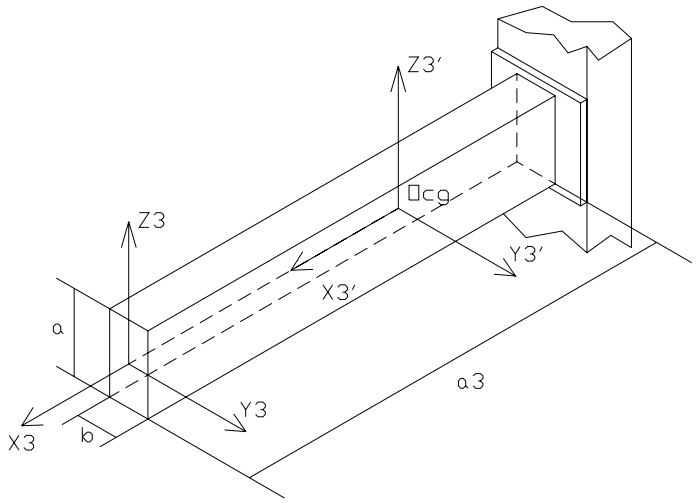


Figure 2-9 Link 3 Center of Gravity

The constructed parameters for link three are:

$$Izz_3 = \frac{m_3}{12}\left((\frac{a_3}{2})^2 + b^2 \right)$$
$$a_3 = 0.4064\ m$$
$$b = 0.0508\ m$$
$$m_3 = 2.0\ kg$$
$$Izz_3 = 0.0073\ kgm^2$$

Plugging the constructed parameters of each link into Equation (2.54), the matrices become:

$$\begin{bmatrix} 0.203+6.8d_2^2 & 0.406 & 0.007 \\ 0.813 & 14.6 & 0 \\ 0.015 & 0 & 4.015 \end{bmatrix} \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{d}_2 \\ \ddot{d}_3 \end{bmatrix} + \begin{bmatrix} 0 & 13.6d_2\dot{\theta}_1 & \dot{d}_2 \\ -6.8d_2\dot{\theta}_1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{d}_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 19.6 \end{bmatrix} = \begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} \qquad (2.64)$$

Simulating Equations (2.63) and (2.64) using Matlab provides the graphs showing the control error and output react to a programmed settling time of 0.5 seconds. The position error graph in Figure 2-10 shows the position error of each axis attenuating to zero in approximately 0.5 seconds.

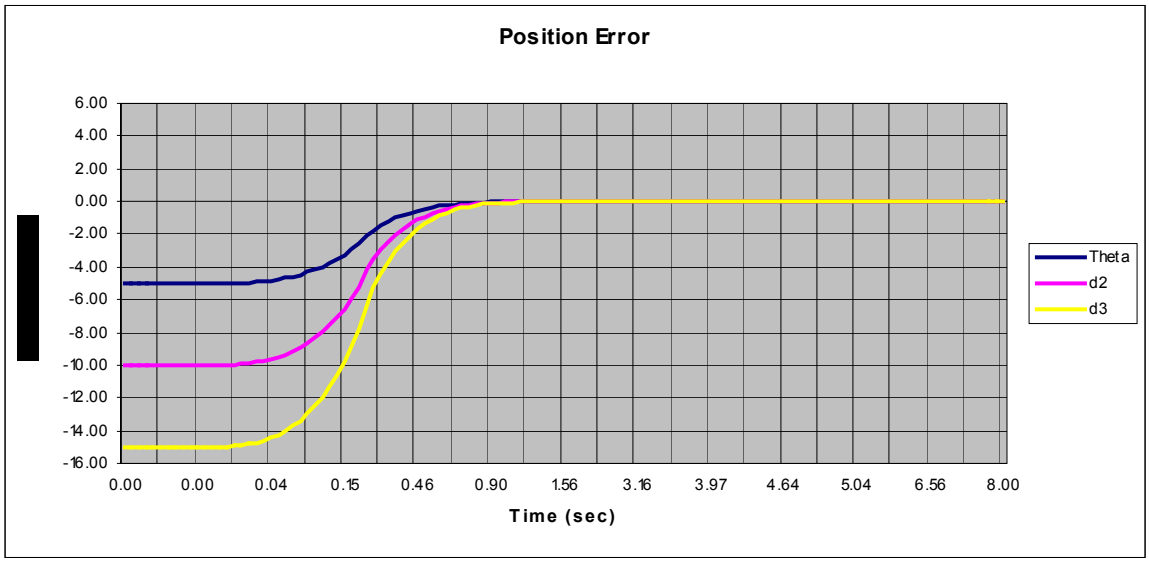


Figure 2-10 Position Error

The velocity error of each axis is illustrated in Figure 2-11.

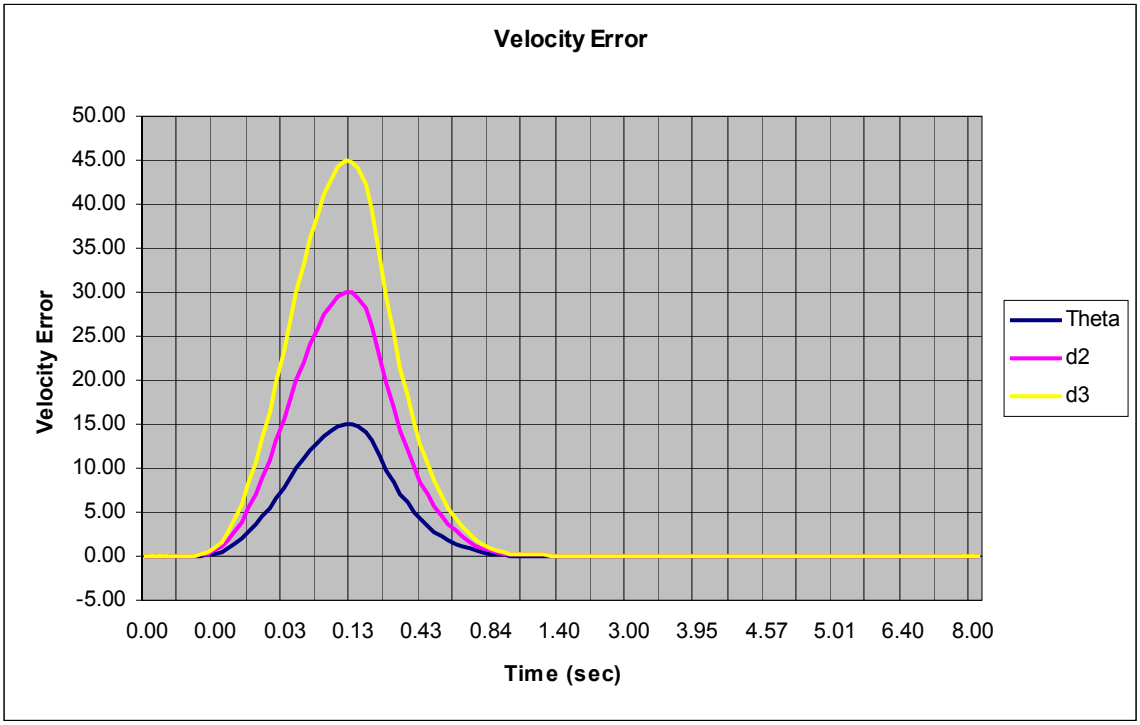


Figure 2-11 Velocity Error

Figure 2-12 represents the desired position trajectories of each axis.

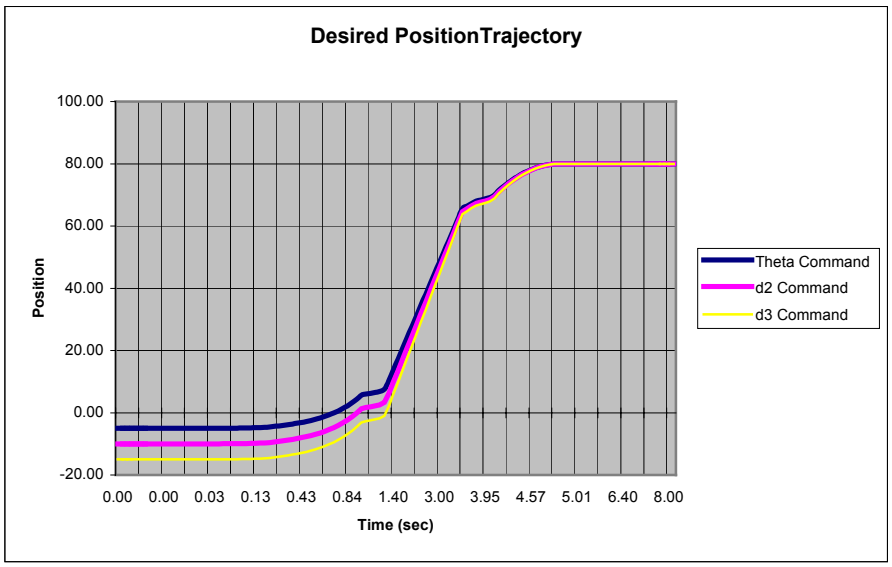


Figure 2-12 Desired Position Trajectory

Figure 2-13 illustrates the actual position simulated results of the PD compensator (Equation (2.63)).

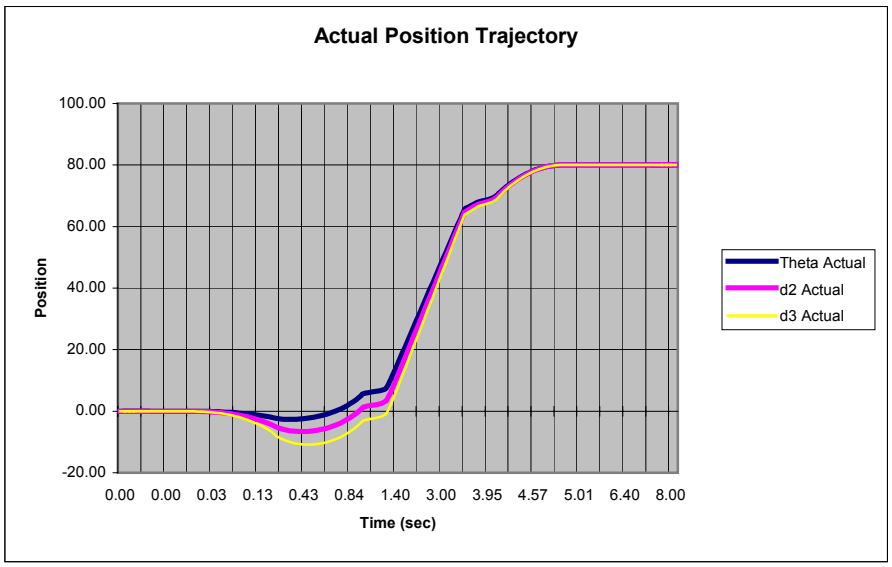


Figure 2-13 Actual Position Trajectory

Figure 2-14 represents the desired velocity trajectories of each axis.

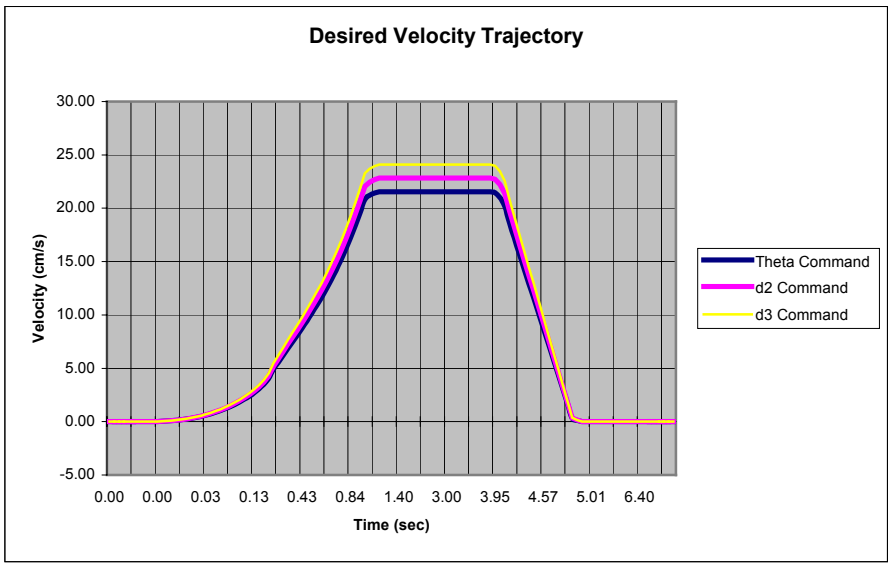


Figure 2-14 Desired Velocity Trajectory

Figure 2-15 illustrates the actual velocity simulated results of the PD compensator (Equation (2.63)).

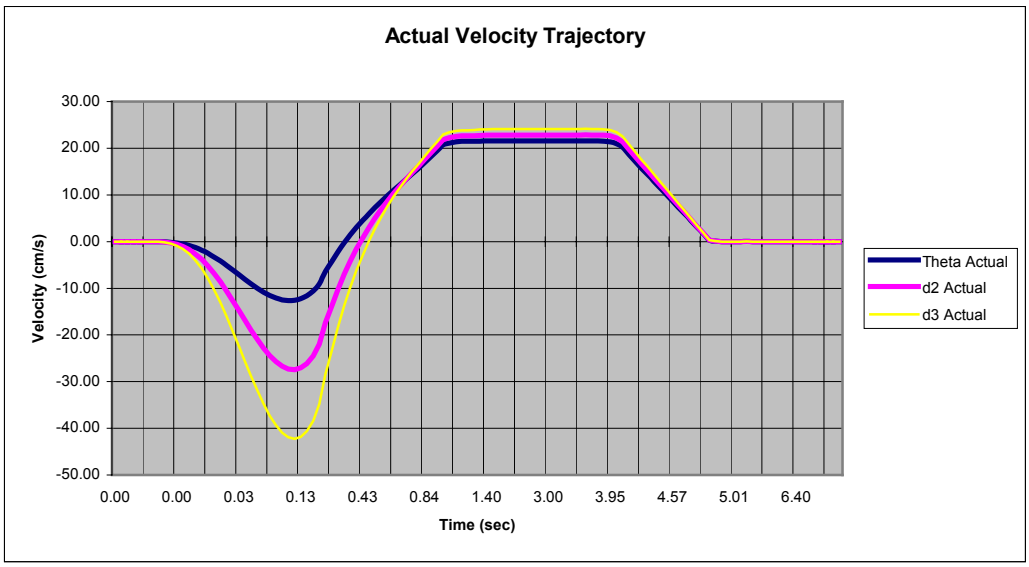


Figure 2-15 Actual Velocity Trajectory

Figure 2-16 represents the desired acceleration trajectories of each axis.

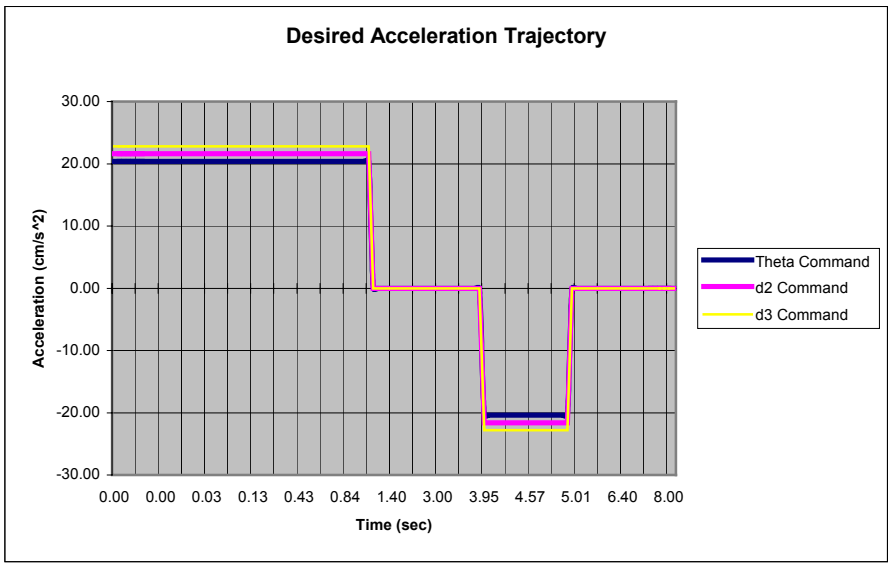


Figure 2-16 Desired Acceleration Trajectory

Figure 2-17 illustrates the actual acceleration simulated results of the PD compensator (Equation (2.63)).

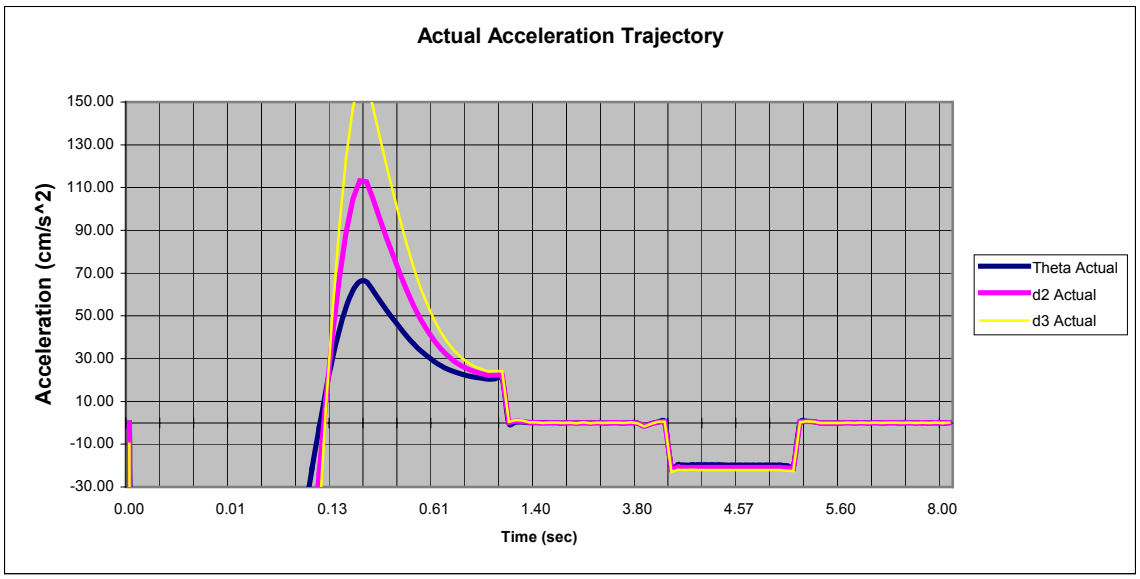


Figure 2-17 Actual Acceleration Trajectory

3 COLOR IMAGE PROCESSING

3.1 Introduction

Prior to the introduction of the microprocessor in the 1970s [7] image data collection and analysis was tedious, unpredictable, and slow. With the introduction of the microprocessor, and arrival of powerful desktop computers, image processing has taken on a whole new dimension, the digital dimension.

Digital imaging has benefited greatly from the invention of the personal computer. Most common households today have some sort of digital imaging device. With the integration of imaging devices, personal computers and processing techniques, new doors have opened for many fields of industries to take advantage of the new technology of image processing. One field of industry taking advantage and taking no limitations is robotics. Robots and "vision" have been conceptualized ever since man conceptualized the human machine. Today the present level of robotic technology in the areas of machine vision is still

primitive compared to the adaptability and ingenuity of the human. Still great strides have been made in this new evolving industry. Robots have successfully been retrofitted with "vision" in assembly manufacturing and other industrial areas since the first robot vision system developed in 1963 [5].

There are two typical setups for a vision-based robotic position and orientation measurement system. One is to fix the camera or cameras to a permanent structure independent of the robotic manipulator. While the robot changes it's configuration, the cameras can view the position and orientation of the end-effector with respect to some calibration point to determine position and orientation. This particular setup generally requires a large field of view, which in turn may reduce measurement accuracy. Higher resolution cameras may be used, but ultimately this will increasing the cost and complexity of the image processing part. The second setup would be to mount the camera or cameras to the end-effector. Now the camera "sees" what the end-effector "sees". In this particular setup the camera does not need a large field-of-view or high accuracy since the cameras only need to perform local measurements. The global information of the robot end-effector's position and orientation may be provided by a stationary calibration fixture [6].

The following sections describe the design and development of integrating a vision feedback motion system into a robotic automation package.

3.2 Color Image Processing and Computers

The manipulation and analysis of pictorial information (image processing) is commonly done by a personal computer.

Before an image can be manipulated, it must be sampled and quantized. A spatially continuous image is typically sampled into a rectangular 2D array containing quantized data. The quantized data commonly referred to as "raw data", represents the image in a number format for easy manipulation by computer algorithms. The fundamental unit of a sampled image is a picture element or pixel [8].

The value of each pixel is equal to the average intensity of the image covered by that pixel [8]. The number of pixels used to store an image is called the resolution. Common resolution sizes used in digital image processing are 256x256, 320x240, and 640x480. Larger resolutions require more memory and longer processing times making them inefficient for a cost affective low detail application, such as the one implemented in this research.

Bits represent pixel values. The number of gray shades determines the number of bits required to determine a pixel value [9]. Black and white images only require 1 bit per pixel (0 or 1) although 8 bits (1 byte) can represent a black and white image with the values being 0 or 255. Grayscale images use 8 bits (1 byte) to represent 256 levels of shade. Figure 3-1 shows the raw data matrix representing a grayscale image.

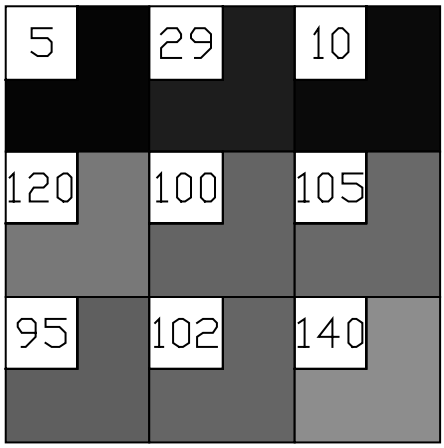


Figure 3-1 Grayscale Pixel Grid

Each pixel (represented by a square on the grid) has a value to determine it's level of shade from black (0) to white (255). This has been accepted as the standard to represent a gray scale image since the human visual system cannot distinguish more than 256 levels of shade [9]. For true color images, 24 bits (3 bytes) represent each pixel. Each byte represents a value for one of the primary colors; red, blue, and green. Figure 3-2

shows the raw data matrix for an RGB 24 bit color picture. Each big square represents a pixel. Each pixel has a 1x3 matrix of red, green and blue to represent a pixel.

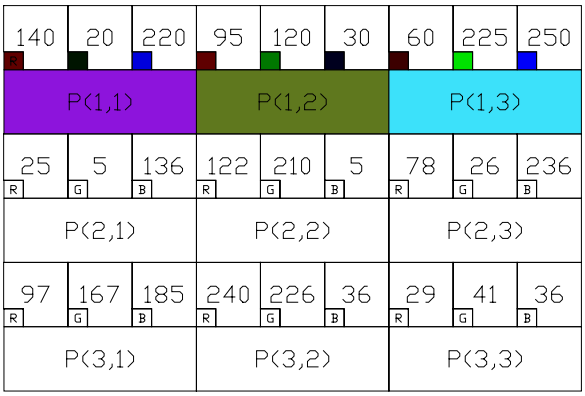


Figure 3-2 Color Pixel Grid

These three colors in different combinations have been determined and shown by James Clerk Maxwell to represent the full visible spectrum of light [8].

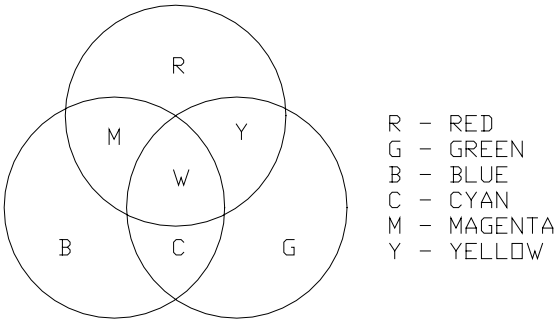


Figure 3-3 Primary Colors

The relationships between the primary colors are shown in Figure 3-3.

3.3 Design and Development

Image capture has many disturbances and unknowns that can make a digital image appear different from capture to capture. One of the easiest ways to help control the changes and variations is to control the light source used to reflect the light off of the surface in focus. Light control is a key aspect for the reproducibility of an object's digital image information. Light is characterized physically by its spectral power distribution (SPD). This characterizes light by the distribution of power as a function of wavelength [10]. Figure 3-4 shows measured relative SPDs for daylight, cool white fluorescent office lighting and an incandescent light.

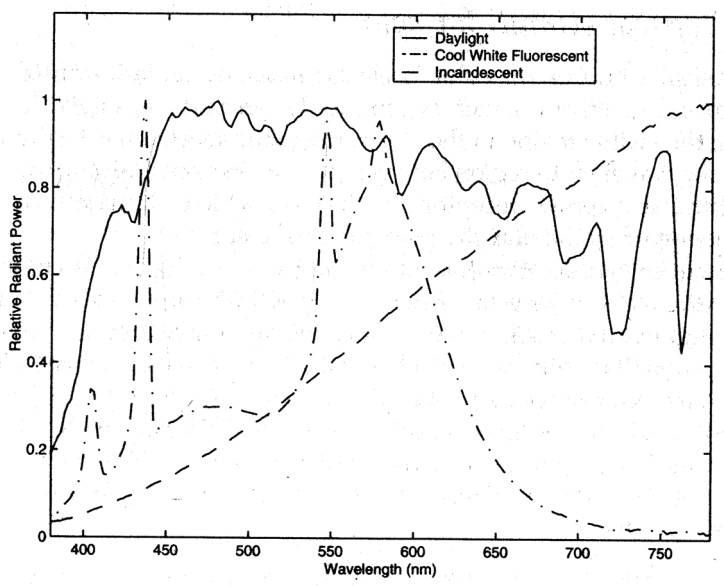


Figure 3-4 Spectral Power Distribution For Visible Light

For this system, fluorescent lighting is used to control the reflected light off of the objects surface. The reason for this choice of lighting is because of the availability of this type of lighting and because of red's relative radiant power reflected under this type of lighting. Red has a wavelength of approximately 650nm, blue is approximately 475nm, and green has a wavelength of approximately 550nm. Therefore, a wavelength of 600 to 700nm (red) has the lowest reflected power.

The process algorithm used to produce the orientation of the part (capacitor) is comprised of using three image transformation techniques. Each transformation may contain multiple steps to produce the desired effect on the image.

1. Reduce the irrelevant information or noise and enhance the image features by applying an image-to-image transformation.
2. Extract the image features that are of interest by applying an image-to-feature transformation
3. Finally, classify the objects of interest and determine the position by performing a feature-to-decision transformation.

Each step is explained further in the following section 3.4 Algorithm.

3.4 Algorithm

The algorithm for the capacitor orientation detection was developed using basic image processing techniques and empirical results that support the basic image processing techniques. The following sections will show the standard techniques along with the empirical data used to implement these techniques.

3.4.1 Image-to-Image Transformation

An image-to-image transformation replaces the original image by a new image [11]. The first image-to-image

transformation applied to the capacitor image will separate each primary color, red, green, and blue, producing three new images. This technique is applied because;

1. The resulting images will be in grayscale allowing the mathematics to become easier by working with three image matrices with only one byte per pixel
2. The red primary image matrix is used to perform the critical image-to-image transformation and image-to-feature transformation.

The reason the red primary image matrix is used to perform most of the critical transformations is because the selection of lighting used to illuminate the object is fluorescent lighting. Thus the reflectance of the red component of the image should not stand out more than the other 2 primary colors, green and blue. This will result in a more robust detection of the image feature being extracted by using the red primary data. By empirically collecting the data, the following results will show this to be true.

Figure 3-5 shows the original image and three new images displayed in gray scale.

Original Red Green Blue

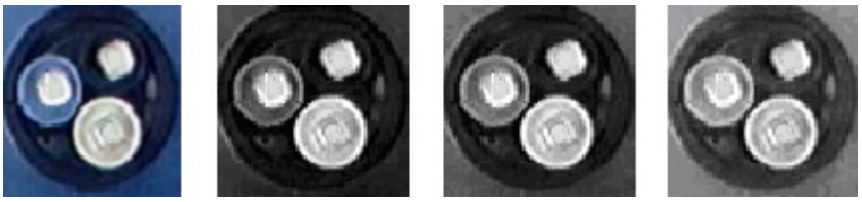


Figure 3-5 Primary Color Images

Each gray scale image is the primary color extracted from the original image.

The algorithm used to perform the primary color separation image-to-image transformation is:

$$\begin{aligned} \text{RedImage}[x][y] &= \sum_{y=j}^{N}\sum_{x=i}^{M}\text{RGBImage}[3x+2][y] \\ \text{GreenImage}[x][y] &= \sum_{y=j}^{N}\sum_{x=i}^{M}\text{RGBImage}[3x+1][y] \\ \text{BlueImage}[x][y] &= \sum_{y=j}^{N}\sum_{x=i}^{M}\text{RGBImage}[3x][y] \end{aligned} \qquad (3.1)$$

This algorithm is developed based on how the computer program stores the image data matrix and how it addresses each element in the matrix.

Referencing Figure 3-5, the red primary image shows to be reflecting the white terminals better against the background. This will benefit the image-to-feature transformation since the white terminals are the feature to be detected.

After the primaries are separated into three matrices, another image-to-image transformation called a smoothing transformation is performed on the red primary image. The smoothing transformation will replace the target pixel's gray

level by a new gray level that is determined by taking the weighted average of it's neighboring pixels (Figure 3-6).

5	29	10	50	250
120	100	105	120	98
95	102	P1 140	111	114
98	80	84	88	93
75	78	72	74	73

Figure 3-6 Smoothing Pixel Transformation

This type of transformation is a spatial operation, since it operates over a 2-d space of numbers. This will eliminate most of the white reflection (noise) received by the camera and blend away any unwanted gray levels in the white terminal (object of interest) areas that are used in the image-to-feature transformation. Figure 3-7 shows the before smoothing transformation and after smoothing transformation of the red primary image.

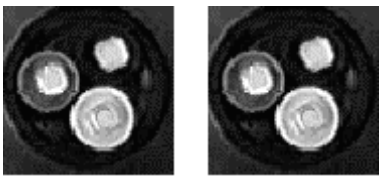


Figure 3-7 Smoothing Transformation

The algorithm used to compute the transformation operates on a 3x3 space. The smoothing spatial operation can be larger or smaller. Larger spatial operations tend to smooth more and loose the detail of the image. Smaller spatial operations will not make much of an impact. A 3x3 spatial operation is a good starting point and usually performs the necessary smoothing to make the required image processing work. Below is the algorithm used to perform the smoothing transformation on the red primary image:

$$\text{RedImage}[i][j] = \left(\sum_{y=j-1}^{j+1} \sum_{x=i-1}^{i+1} \text{RedImage[x][y]} \right) / 9 \qquad (3.2)$$

From Figure 3.6, pixel P1's new value would become:

$$\text{RedImage}[3][3] = 975/9 \cong 108$$

After the image-to-image transformations are complete, an image-to-feature transformation is applied. The feature to be extracted from the image is the four blade white terminal connection located in the center of each colored insulator.

The detection of the terminals are chosen specifically because of the following reasons:

- Since the camera used is a low cost, low-resolution web camera, pixel information is not repeatable and patterns are hard to detect making decision techniques difficult. The terminals reflect the brightest gray levels and are easily detectable and almost always guarantee a distinct pattern.
- The middle of the terminal is important for determining the rotation angle necessary to bring the orientation of the capacitor to the correct location.

The following section describes the techniques and algorithms used to perform the image-to-feature transformations.

3.4.2 Image-to-Feature Transformation

Typical image-to-feature transformations are gray level scaling transformations. In these types of transformations, the gray level in the transformed image depends only on the gray level of the same pixel in the original image [11]. A common gray level scaling transformation is the threshold transformation, which takes the value of a pixel and changes it's value to a lower or upper limit based on the threshold value.

The algorithm to perform the threshold transformation is:

$$i_2(x,y) = \begin{cases} 0 \ \ if \ i_1(x,y) \le T \\ 255 \ \ if \ i_1(x,y) > T \end{cases} \tag{3.3}$$

A histogram is constructed to determine the threshold value T. One way to choose T is to find the minimum point between the two largest peaks. This value will equal the threshold. Estimating based on Figure 3-8, the minimum chosen value that occurs between pixel value 21 and 255 is T=130.

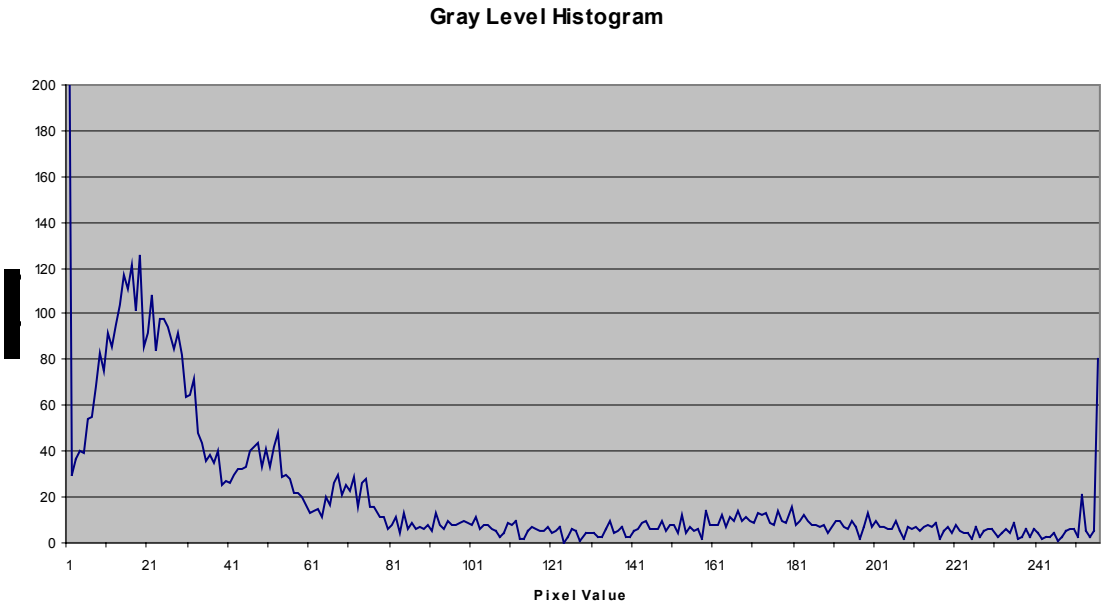


Figure 3-8 Gray Level Histogram for Red Primary

The new image created after the threshold transformation is illustrated in Figure 3-9.

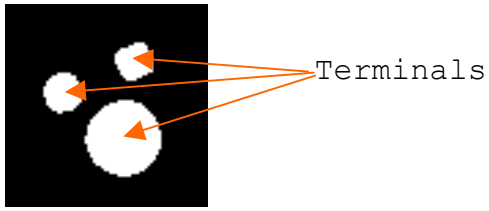


Figure 3-9 Threshold Image Transformation

As illustrated in Figure 3-9, the objects of interest (terminals) are easily determined after the threshold transformation extracts them out. The larger white circle also captures the white insulator surrounding the terminal. The terminals are now used to determine the center point of the

insulator of interest. With the threshold image transformation complete, the next step is to perform an Image to Decision Transformation.

3.4.3 Image to Decision Transformation

Once the features of the image are extracted and easily detectable, a decision has to be made on the features of interest. The color insulators that surround each terminal are one of the features of interest. To determine the color insulator that surrounds a terminal, first a spatial operation is performed to locate each terminal. Using the resulting image after the threshold transformation, the image is horizontally scanned (note the current position as the marker) starting from the first pixel position (top left of the image). The marker moves in step sizes of three. Each group of three pixels are averaged and compared to 255 (three white pixels in a row). Once the average is equal to 255, a vertical scan is applied to average three adjacent vertical pixels. If the value of the three vertical pixels is 255, than the decision is made that a terminal has been detected.

The next step after the terminal has been detected is to determine the selected color insulator around the terminal. The

first pixel that was found to be white during the horizontal scan is the edge of the terminal and insulator. The marker is moved to this position. Next the marker will move three horizontal pixels to the left and then three vertical pixels up. A 3x3 average spatial operation is performed around the marker's new position using Equation (3.4).

$$Avg = \left(\sum_{y}^{y+3} \sum_{x}^{x+3} image[x][y] \right) / 9 \tag{3.4}$$

If the average of the spatial operation is between the selected color insulators limit settings, then a decision is made that the selected color has been located. Table 3-1 below shows the color ranges of each primary color data matrix for each insulator.

Table 3-1 RGB Threshold Values

	Red Image Matrix	Green Image Matrix	Blue Image Matrix
Blue Insulator	55-113	81-137	120-180
Black Insulator	0-70	0-70	0-70
White Insulator	255	N/A	N/A

The ranges were calculated using the data obtained from each primary color image. Each primary color image's raw data was extracted and examined to calculate the average of each insulator color for each primary color matrix.

The blue insulator's average in the red primary data matrix is 84. The standard deviation is 29. Using one standard

deviation from the mean produced the range used to detect the blue insulator in the red primary data matrix. The blue insulator's average in the green primary data matrix is 109. The standard deviation is 28. One standard deviation from the mean produced a range of 81-137. The blue insulator's average in the blue primary data matrix is 150. The standard deviation is 30. One standard deviation form the mean produced a range of 120-180. The same procedure was implemented for the black insulator. The results produced a common 0-70 value range to detect the black insulator. The white insulator was detected differently. After the threshold transformation is completed, the white insulator stands out immediately from the other white circles produced by the terminals. This is evident by the large diameter of the white area in the image, which concludes this terminal to have the white insulator surrounding it.

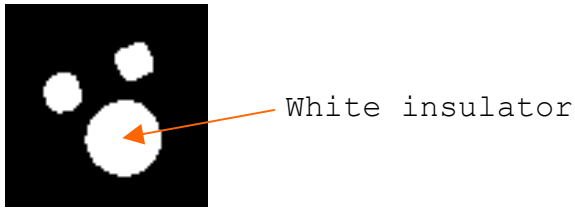


Figure 3-10 White Insulator

By empirically collecting data for each of the primary color images, a mean and deviation was calculated to give a range that is used to determine what color insulator surrounds

the terminal as shown in Table 3-1. This procedure was repeated for three images captured using the standard web camera. The results were accurate and repeatable, which concludes that the procedure is robust.

4 PROCESS DESCRIPTIONS

The following process descriptions summarize the actual operation of the automated subroutines for the Autonomous Robotic Automation System. Each process is broken down into individual tasks that must be performed to successfully operate the robotic system. The Main Process is comprised of multiple subroutines to achieve the autonomous operation of the system.

4.1 Calibration

Calibration establishes an accurate motion profile for all axes to pick up the capacitor from an absolute zero reference point, and determines color selection and orientation of the capacitor. Calibration reduces production error from job to job, by maintaining consistency.

A calibration routine contains a test target (capacitor), and a software routine that the operator will interact with to perform the calibration routine. The following procedure outlines the sequence of events that must take place to ensure a successful calibration.

1. Load the capacitor into the boat and position boat on the detect line.
2. From the HMI main menu (reference section xxx), enable all axes by clicking on the “Enable Axis” check box for each axis.
3. Select “Tools>Calibrate”.
4. From the “Align” tab, click the “Camera View” button.
5. Using the align jog buttons, position the gripper fingers so they will grip the top cap of the capacitor. Once they are in position press “Set Position 1” on the HMI.
6. Using the “Jog” buttons, jog each axis so the capacitor is in the center of the camera’s Field-Of-View (FOV). Once the capacitor is roughly aligned in the camera’s FOV, press “Preview Position”.
7. A circle will be imposed on the image. The capacitor needs to be positioned in the circle with the edges of the capacitor in the circle.
8. Press “Camera View” to view the live image. Continue to jog axes and “Preview Position” until the capacitor edges align in the circle. Once the capacitor is positioned correctly in the camera FOV, press “Set Position 2” and focus the camera.

9. To calibrate the system to detect a color and position the orientation of the capacitor, click the “Color Select” tab.
10. Rotate the capacitor positioning the correct color insulator in the desired position.
11. Click the “Capture Image” button. An image will be captured and displayed on the HMI.
12. From the “Select Color” pull down combo box, choose the desired color. If the color detection routine works correctly, a red crosshair will be positioned in the center of the selected insulator.
13. Save settings by pressing the “Save Settings” button on the HMI. The position for each move will be displayed on the HMI.

4.1.1 Calibration Flowchart

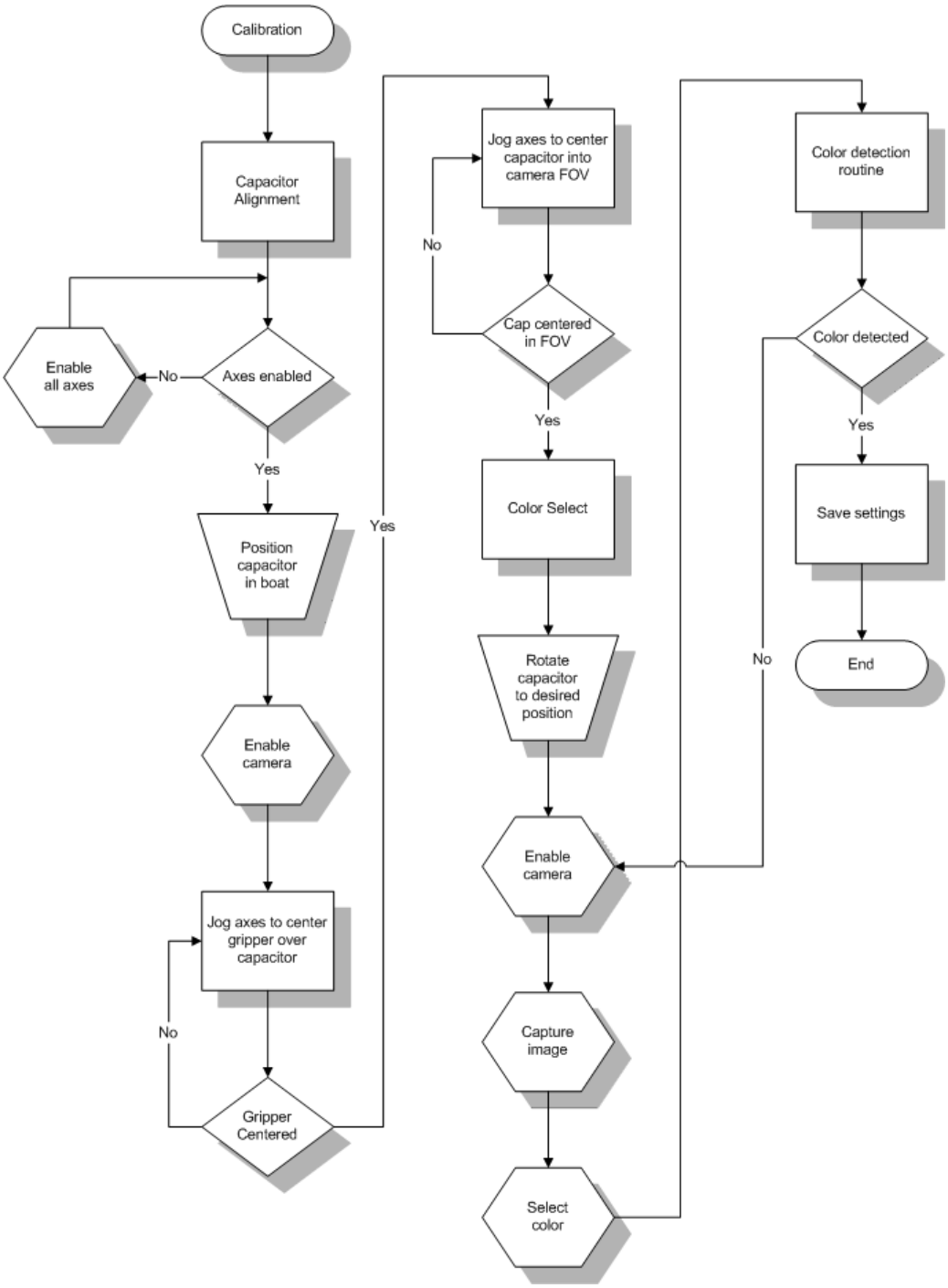


Figure 4-1 Calibration Flowchart

4.2 Capacitor Orientation

The Capacitor Orientation routine detects the current position of the capacitor and determines what rotation angle needs to be applied to rotate the capacitor to the correct orientation. The orientation of the capacitor is based upon the selected color insulator from the calibration routine (reference section 4.1 Calibration).

This Capacitor Orientation routine runs in the background during the Main Process run. After a part is detected, the Capacitor Orientation routine executes.

The initial image captured has a 320x240 resolution. The area of interest is a 74x74 frame that is called the capacitor frame. This frame begins at pixel 123x83 (reference Figure 4-2). Once the image is captured the capacitor frame image is extracted and stored in an image matrix. The remaining pixels outside the capacitor frame are ignored. The top left corner of the 74x74 capacitor frame is defined as the origin (reference Figure 4-2).

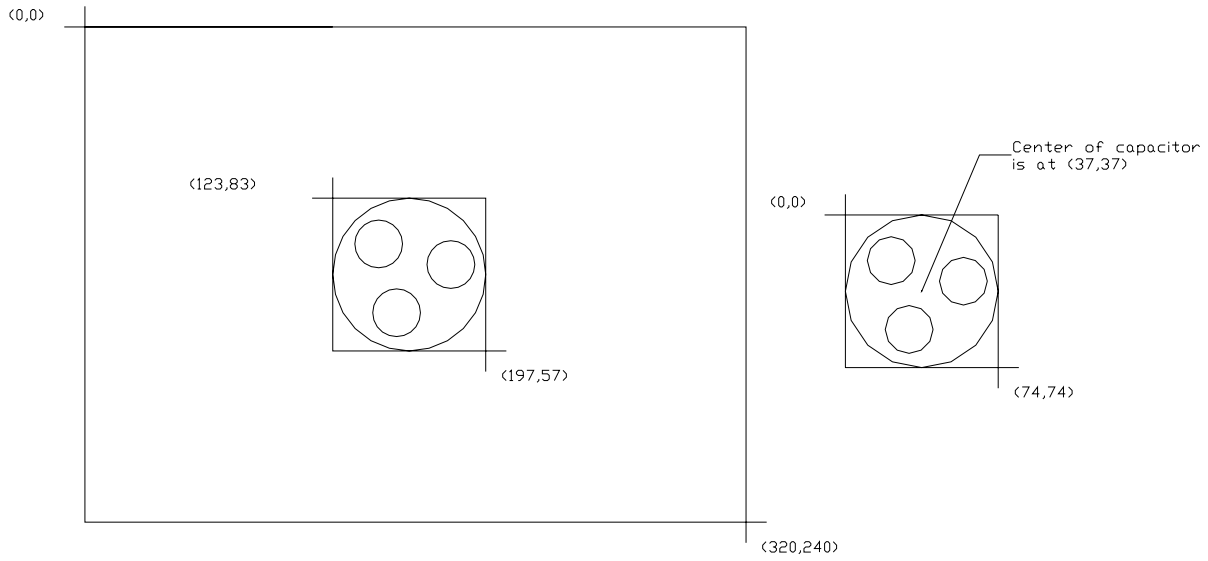


Figure 4-2 Image Pixel Grid

After a successful calibration routine (reference section 4.1 Calibration), the center of the capacitor must be located at the center of the capacitor frame (37,37). This is very important for the calculation of the angle of rotation to be accurate.

The first step in the Capacitor Orientation angle calculation is to define the insulator's terminal center point (X,Y) (reference Figure 4-3) with respect to the origin. This procedure is executed during the calibration procedure and when part detection occurs during the main process.

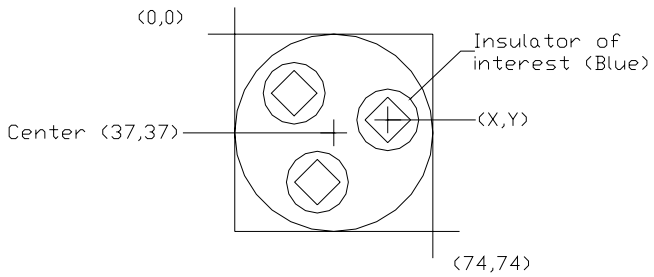


Figure 4-3 Capacitor Area of Interest

To calculate the center point of the terminal a scan routine is run around the edge of the terminal to determine the horizontal and vertical major axes. The black and white threshold transformed image is used in this routine to extract the terminals and calculate the center of the terminal of interest.

The following steps outline the algorithm used to determine the horizontal and vertical major axes.

1. First, the far most right white pixel is located by scanning along the edge as shown illustrated in Figure 4-4. Figure 4-4 represents each white pixel element of the terminal that was captured by the camera. To determine the last white pixel, each pixel below the current pixel is examined to see if it is black. If it is not black the marker indexes to the right. If it is black, the marker will index down one unit. It will continue to index down until a white pixel is found or

until ten indices. If the marker indexes 10 times before finding a white pixel, then the last indexed unit to the right is considered to be the far most right white pixel. After the last white pixel on the right has been detected, the horizontal location of that pixel (x) minus two units is the saved far right pixel position.

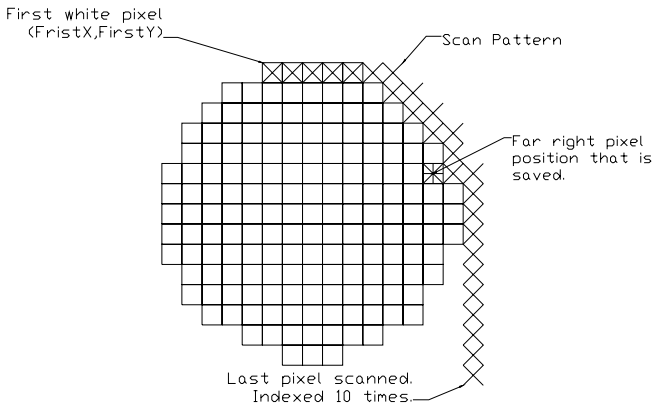


Figure 4-4 Far Right Pixel Scan

2. After the far right pixel location has been saved, a scan run along the left side of the terminal is performed to locate the far most left pixel in the same routine that the far most right pixel was located. After the last white pixel on the left has been detected, the horizontal location of the pixel (x) plus two units is the saved far left pixel position.

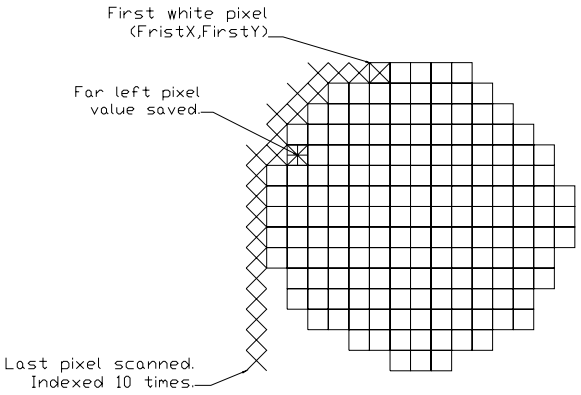


Figure 4-5 Far Left Pixel Scan

3. A bottom scan is performed to determine the last pixel position located at the bottom of the terminal.

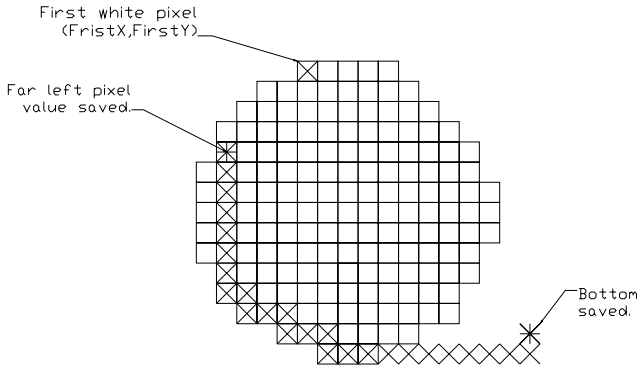


Figure 4-6 Bottom Pixel Scan

4. After the edge pixel positions are saved, a horizontal and vertical line is defined. The midpoint of each line is determined to be the midpoint of the terminal.

(4.1)

$$X_{mid} = \frac{X_{FarRight} - X_{FarLeft}}{2}$$
$$Y_{mid} = \frac{Y_{Bottom} - Y_{First}}{2}$$

The absolute midpoint position of a terminal is:

$$X_{Abs} = X_{mid} + X_{FarLeft}$$
$$Y_{Abs} = Y_{mid} + Y_{First} \qquad (4.2)$$

Once the absolute midpoint position of the terminal is calculated, the angle of rotation algorithm below calculates the necessary rotation to bring the capacitor to the correct orientation.

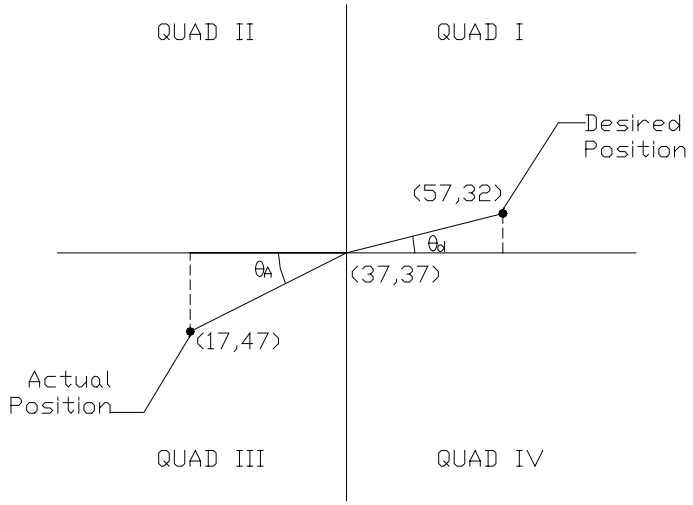


Figure 4-7 Orientation Example

Figure 4-7 demonstrates an example for the algorithm used to calculate the angle of rotation.

The following algorithm calculates the rotation angle necessary to bring the actual position of the terminal to the desired position.

Let the desired terminal position be denoted as:

$$X_{AbsD} \ \& \ Y_{AbsD}$$

Calculation of the desired angle θ_{des}:

Quad I

$$X = X_{AbsD} - X_{center}$$
$$Y = Y_{AbsD} - Y_{center}$$
$$\theta_d = tan^{-1}(\frac{Y}{X})$$

From the Figure 4-7:

$$X = X_{AbsD} - X_{center} = 57 - 37 = 20$$
$$Y = Y_{AbsD} - Y_{center} = 37 - 32 = 5$$
$$\theta_d = tan^{-1}(\frac{5}{20}) = 14.0^0$$

Quad II & III

$$X = X_{AbsD} - X_{center}$$
$$Y = Y_{AbsD} - Y_{center}$$
$$\theta_d = tan^{-1}(\frac{Y}{X}) + 180^0$$

Quad IV

$$X = X_{AbsD} - X_{center}$$
$$Y = Y_{AbsD} - Y_{center}$$
$$\theta_d = tan^{-1}(\frac{Y}{X}) + 360^0$$

The same procedure follows to calculate the actual terminal angle θ_a. Using Figure 4-7 as an example the actual terminal angle located in quadrant III is:

$$X = X_{AbsA} - X_{center} = 17 - 37 = -20$$
$$Y = Y_{AbsA} - Y_{center} = 37 - 47 = -10$$
$$\theta_a = tan^{-1}(\frac{Y}{X}) + 180^0 = tan^{-1}(\frac{-10}{-20}) + 180^0 = 206.6^0$$

Delta θ is equal to:

$$\theta_{Delta} = \left|\theta_{AbsA} - \theta_{AbsD}\right| = 192.6^0$$

The gripper axis will rotate counterclockwise if the delta angle of rotation is greater than 180 degrees. If the angle of rotation is less than 180 degrees, the gripper axis will rotate clockwise by θ_{Delta}.

For the example, the actual angle of rotation for the gripper axis is $\theta_{CC} = 360^0 - \theta_{Delta} = 167.4^0$ counter clockwise

4.3 Color Detect

The Color Detect routine detects the color insulator located around the terminal. This routine is initiated after the part has been detected by the sensor. The following steps list the procedure used to determine the color of the insulator around the terminal.

1. The first step is to locate the terminal. This is accomplished by scanning a black and white 74x74 image of the capacitor from left to right. While scanning, three horizontal adjacent pixels are added to see if all three are white pixels.
2. Once three horizontal adjacent pixels are white, a vertical scan is performed. This scan will continue down until no more white pixels are found. If the scan returns a value greater than five vertical adjacent pixels, then it is assumed that a terminal is located.
3. The last step is to perform an average of nine pixels outside of the terminal (3x3 matrix). The average value is compared to the RGB threshold values for each color to determine the color surrounding the terminal (reference Table 3-1).

4.3.1 Color Detect Flowchart

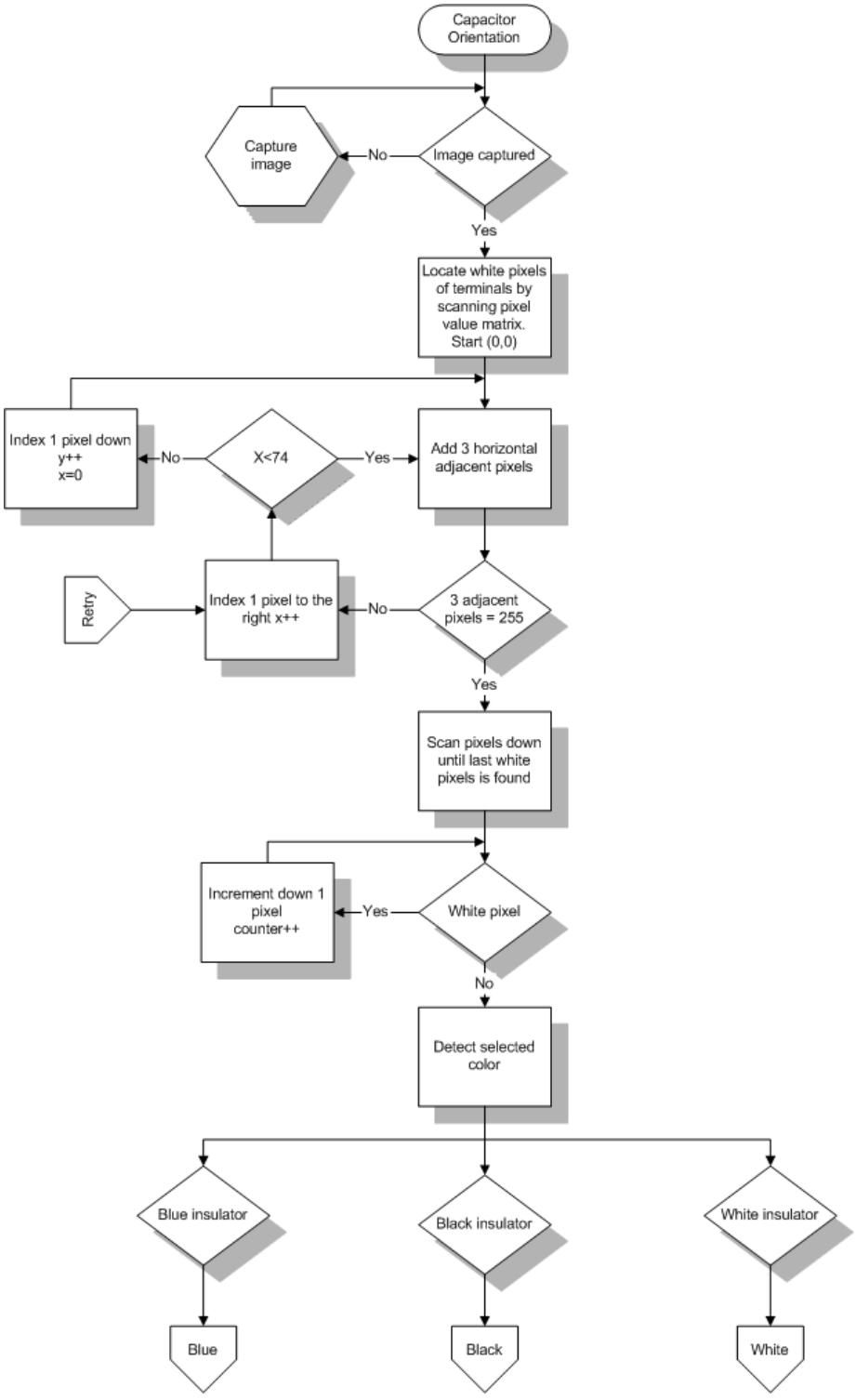


Figure 4-8 Color Detection Flowchart

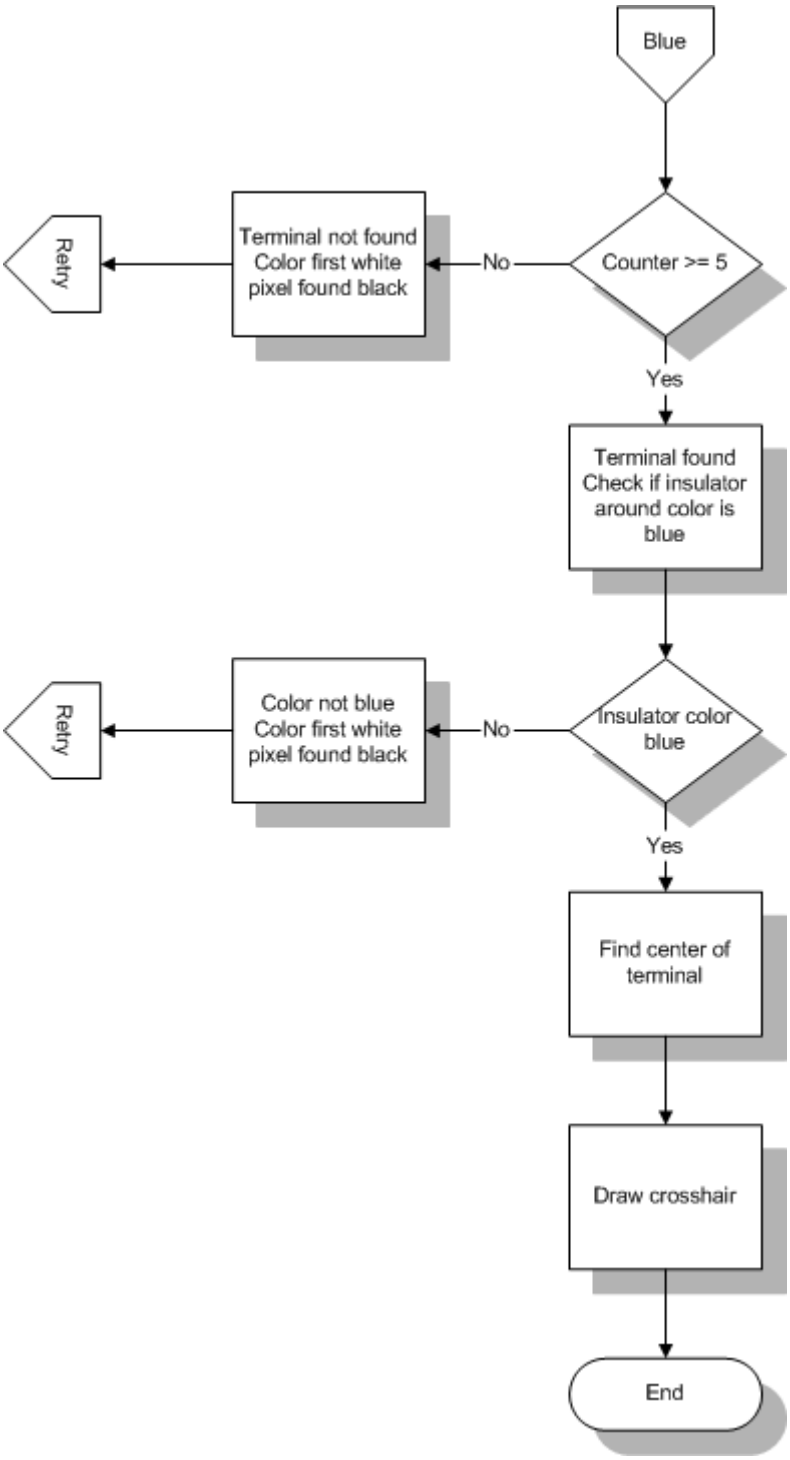


Figure 4-9 Blue Color Detection

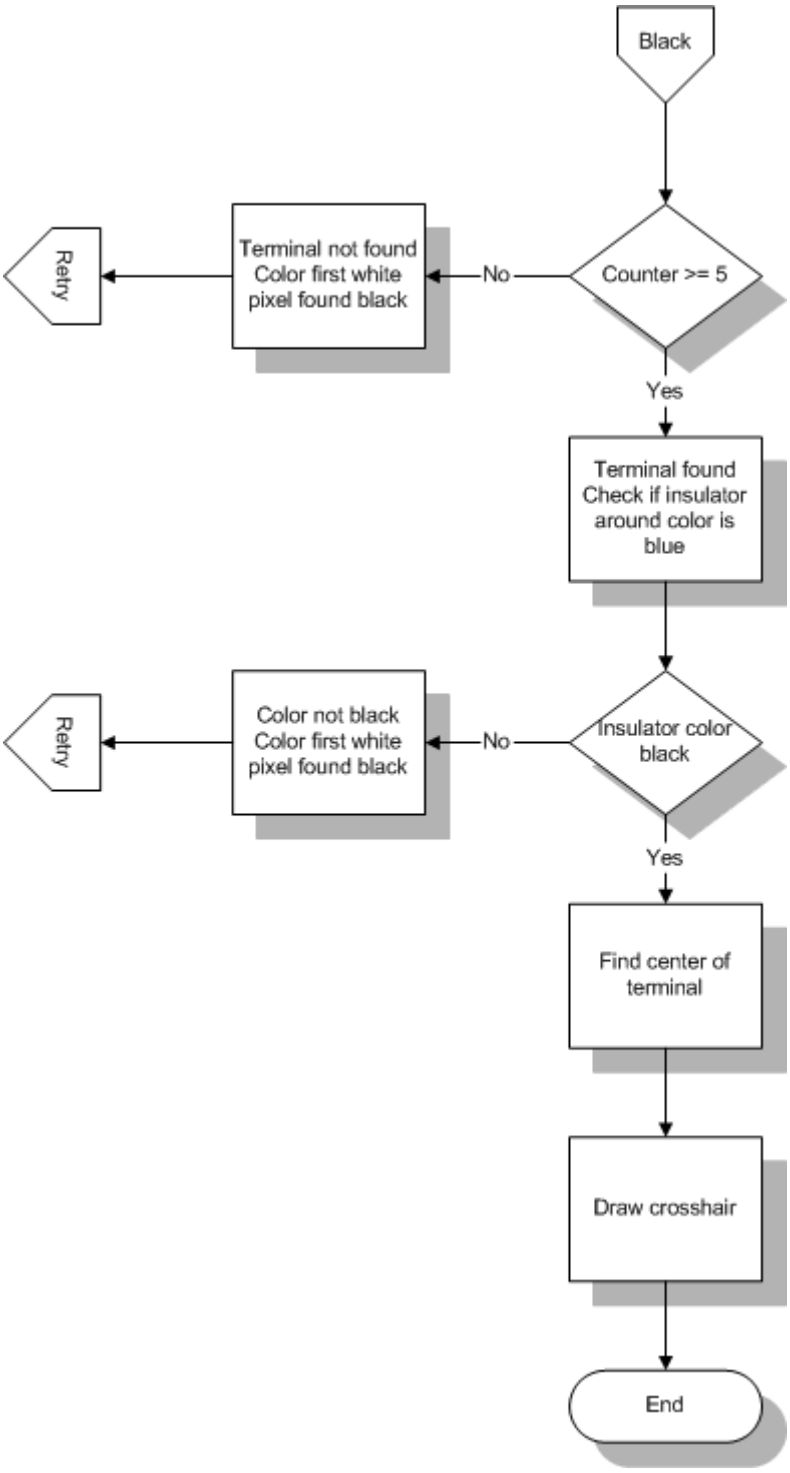


Figure 4-10 Black Color Detection

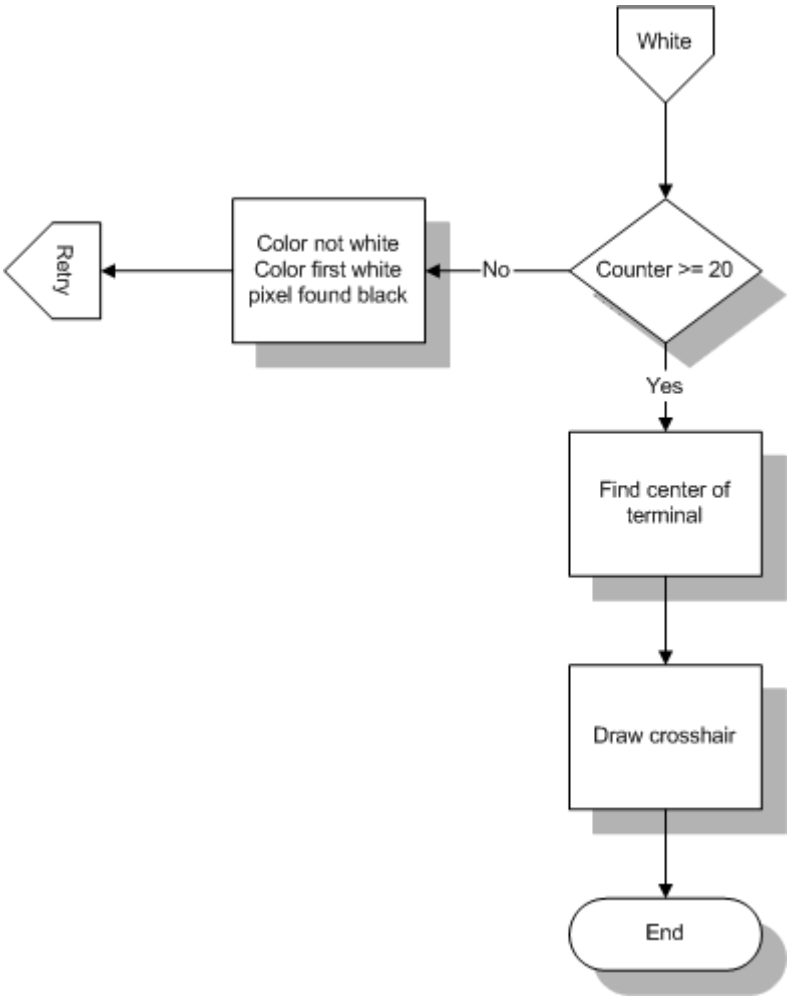


Figure 4-11 White Color Detection

4.4 Motion

The motion routine executes independent motion for four axes; horizontal, rotation, vertical, and a revolute gripper. Each axis has a programmable velocity, accel/decel, and position value to define a motion profile. Up to four sequential moves can be programmed for each axis. Coordinated motion can be achieved by programming the order of moves for each axis. After a motion profile is programmed, the operator will have the option to repeat an individual axis, or all axes by checking one of the "Repeat Motion" check boxes on the HMI.

Each axis has a home limit switch. The vertical and horizontal axes have end travel limit switches. All motions are absolute moves from a zero reference point (usually home). For example, if the axis encoder has been zeroed, and a move to the 10cm position is entered, the axis will move to the 10cm mark (based off of encoder counts). If the next move commands the axis to move to position 12cm, the axis will move 2cm from the 10cm mark. If the axis was programmed to execute relative moves, the axis would have moved 12cm from the 10cm mark to put it at the 22nd cm position.

The following procedure lists the necessary steps to create and run a successful motion profile.

1. Enable all axes.
2. Each axis must be initialized to ensure the encoders are reset at the first index pulse detected after the home position is located. From the HMI select "Motion>Initialize Board>Yes" to execute the initialize routine.
3. For each axis that is part of the motion profile, program the velocity (cm/s for linear and deg/s for rotation), accel/decel (cm/s^2 for linear and deg/s^2 for rotation), and a position (cm for linear and deg for rotation). If an axis is not part of the move, leave the position parameter blank. If an axis is not part of the motion profile, disable the axis by removing the check from the "Enable Axis" check box.
4. Select the order for each move or leave blank. Each axis will start to move at the same time if the order for each axis is not selected.
5. Press the "Enter" button to store move parameters. Select the "Next" button if another move is required.
6. Repeat steps three through five until the desired number of moves up to four maximum are programmed.
7. Select which axis moves are to be repeated.
8. Review motion profile.

9. Once motion profile is approved press “Run” to execute the motion profile.

Once the “Run” button is pressed, the axis will begin motion and an “Axis Status” dialog will pop-up displaying the axis velocity and position. The motion can be halted and continued after a halt.

4.5 Main Process

The Main Process incorporates the above subroutines into a fully autonomous robotic automation system with vision feedback. The following steps list the procedure necessary to prepare the system to run a main process.

1. Initialize the axes by going to “Motion>Initialize Board>Yes”.
2. After initialization is confirmed successful, enable all the axes.
3. Run the calibration routine (reference section 4.1 Calibration).
4. After the calibration routine is completed, enter in the velocity and acceleration values for each axis and move that will run for the main process (reference section 4.4 Motion).

5. Confirm all the setpoints. Press the "Run" button to execute the Main Process.

The following sequence of events will execute when the Main Process is executed.

1. The system will wait until a boat with a part has been detected by the proximity switch (software button simulates proximity switch).
2. Once the switch detects that a part is in place, the camera will take a snap shot of the part.
3. The image is captured, and the color detection routine will execute to find the color of interest.
4. The Capacitor Orientation routine will execute. The rotation angle for the gripper is calculated.
5. The motion routine is called to run after the orientation rotation angle is calculated. The system will pick up the capacitor and perform the move sequences.
6. The system will repeat the above steps every time the proximity sensor detects a part present.

5 DESIGN AND DEVELOPMENT

This chapter describes the hardware and software tools used in the development of the autonomous robotic automation system with vision feedback. The hardware and software have been chosen to provide a seamless integration for this hybrid system. The following hardware and software sections identify the various components and their function.

5.1 Hardware

The main hardware components that make up the hybrid system are:

- NI's PCI-7344 motion controller
- Commercial PC
- Advanced Motion Control Amplifiers
- Power Supplies
- Lintech Manipulator
- Galil Servo Motors
- Vision System
- Gripper System

5.1.1 Motion Controller PCI-7344

National Instruments PCI-7344 motion controller is a true motion control computer that operates in a host PC environment as a motion coprocessor. The controller has the ability to control repetitive motion and take measurements simultaneously.

The advanced motion architecture controls up to four servo axes with quadrature, incremental, or single-ended encoder inputs. Circular, spherical, helical interpolation and blended-motion profiles provide infinite trajectory control.

5.1.2 Commercial PC

The commercial desktop PC hosts the motion controller card and the HMI to comprise the Supervisor Control and Data Acquisition (SCADA) system. The PC is a Gateway PC with a Pentium 3 1 GHz processor, 256 MB of RAM, and a 30 GB harddrive.

5.1.3 Amplifiers

The drivers for the servo axes are Advanced Motion Controls 12A8 servo amplifiers. These small size PWM servo amplifiers

are designed to driver brush type DC motors at a high switching frequency.

12A8 model specifications

- DC supply voltage: 20-80 VDC
- Peak Current (Two seconds max): 12 Amps
- Maximum continuous current: 6 Amps
- Switching Frequency: 36 kHz

5.1.4 Power Supplies

The system utilizes two power supplies. One unregulated power supply feeds all amplifiers with 20 amps at 80 VDC. A second 200 watt 24 VDC power supply feeds the interface electronics, which have voltage regulators to provide the necessary input power requirements for the miscellaneous low voltage electronics.

5.1.5 Lintech Manipulator

The positioning system consists of a Lintech 300K theta axis, and two 100K series linear axes. One linear axis provides X travel, and the other linear axis provides Z travel.

Theta Axis

- 6” diameter standard grade rotary table
- 180:1 worm gear
- Reed home switch

X-Axis

- 16 inches of travel
- 5 pitch non-preloaded rolled ball screw
- Mechanical limit and home switches

Z-Axis

- 16 inches of travel
- 5 pitch non-preloaded rolled ball screw
- Mechanical limit and home switches
- 24VDC fail safe brake

5.1.6 Galil Servo Motors

Galil's N23-53-1000 NEMA 23 brush-type servo motors drive the Lintech position system. Each motor is equipped with a 1000 pulse per revolution differential quadrature encoder with a maximum output frequency of 100 kHz. The motors can provide up to 53 oz-in of continuous torque and 300 oz-in of peak torque at a maximum angular velocity of 6500 rpm.

5.1.7 Vision System

The vision system consists of a D-Link DSB-C300 USB webcam. The D-Link DSB-C300 webcam is a High Resolution USB Digital Video Camera. The DSB-C300 supports video with 64-Million colors at 30 Frames per second by using the high bandwidth of USB, and advanced light-sensing CMOS. technology. As a fully contained

unit, no video capture card or additional hardware is necessary. The unit simply attaches via a standard USB port.

5.1.8 Gripper System

The gripper system is comprised of HD System's RH-11D-6001 high precision DC servo motor with incremental encoder, and Techno-Sommer's GED1306 3 jaw electric gripper. Each jaw has a 6 mm range of motion. The motor has a continuous torque equal to 311 oz-in at maximum angular velocity of 100 rpm. The encoder has 1000 pulse per revolution.

5.2 Software

This section reviews the software tools used to develop the HMI system that controls the autonomous robotic automation system with vision feedback.

5.2.1 Borland C++ Builder 5.0

C++ Builder is a powerful ANSI C++ development environment for rapidly building applications. The integrated development

environment (IDE) provides all the tools needed to design, develop, test, debug, and deploy applications.

C++ Builder's development environment includes a visual form designer, Object Inspector, Component palette, Project Manager, source code editor, debugger, and installation tool. Moving from the visual representation of an object (in the form designer), to the Object Inspector to edit the initial runtime state of the object, to the source code editor to edit the execution logic of the object is made effortlessly.

C++ Builder 5.0 IDE developed the Human Machine Interface (HMI) used to control and monitor the autonomous robotic automation system with vision feedback.

5.2.2 Adobe Photoshop

Adobe Photoshop is a powerful graphic design software package. This software helped develop the image processing algorithms used to detect the orientation of the capacitor by opening the images captured by the D-Link webcam and analyzing them.

5.2.3 Measurement & Automation Explorer (MAX)

MAX provides direct access to the motion controller PCI-7344. MAX makes it easy to configure the motion controller, add new channels, execute system diagnostics, and view devices and instruments connected to the system.

5.2.4 Human Machine Interface (HMI)

The Human Machine Interface (HMI) is the PC based operator panel that is used to operate the ARASV. The HMI system provides several features for the efficient operations of the machine. The primary features of the HMI system are described in the following sections.

5.2.4.1 Main

The Main screen shown in Figure 5-1 is the first screen that appears when the program is executed. The Main screen allows the user to program a motion profile, run a main process, and navigate to other functions of the HMI. This section defines the functions of this screen.

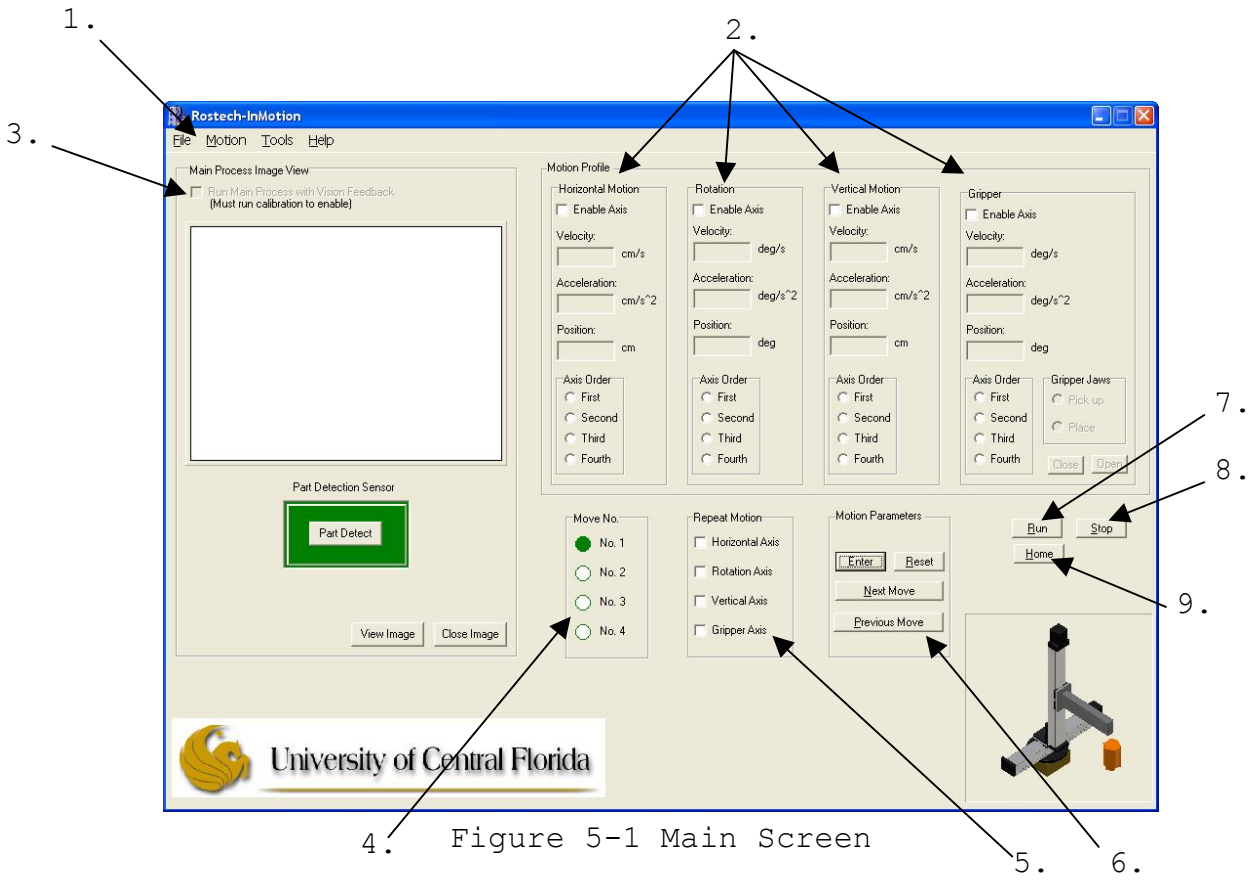


Figure 5-1 Main Screen

1. The main menu; "File", "Motion", and "Tools".
   a. File
      i. Open: Allows the operator to open a saved motion profile.
      ii. Save As: Allows the operator to save a programmed motion profile.
      iii. Exit: Closes video driver and exits the program.
   b. Motion
      i. Initialize Board: Executes the initialize routine. The initialize routine will map the axis resources, load the gain parameters, home all axes and reset the position.

ii. New Trajectory/Reset: Allows the operator to erase the current motion profile and start a new trajectory.

iii. Enter: Saves the current "Move No." motion parameters and allows the operator to click "Next Move".

iv. Run: Executes the current programmed motion profile.

c. Tools

i. Camera: Enables the operator to format the video image and set the video image settings.

ii. Calibrate: Opens the "Calibrate" dialog.

2. Defines the motion profile settings for each axis. Reference Figure 5-2 and Figure 5-3.

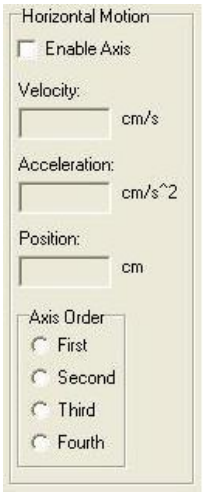


Figure 5-2 Axis Motion Parameters

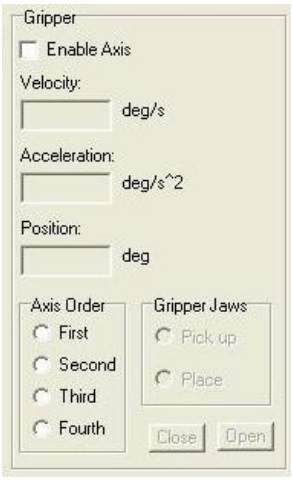


Figure 5-3 Gripper Motion Parameters

a. Enable Axis: Enables the axis.
b. Velocity: Input box for velocity setpoint.
c. Acceleration: Input box for accel/decel setpoint.
d. Position: Input box to define end-point position with respect to the zero reference point. All moves are absolute moves with respect to the origin.
e. Axis Order: Selects the order with respect to the other axes for which the axis will execute its move to the programmed position when the "Run" button is pressed.
f. Gripper Jaws: Selects whether the gripper jaws will pick up (close) the object at the end of a move or place (open) the object at the end of a move.
g. Gripper Close: Manually closes the gripper jaws.
h. Gripper Open: Manually opens the gripper jaws.

3. Image View

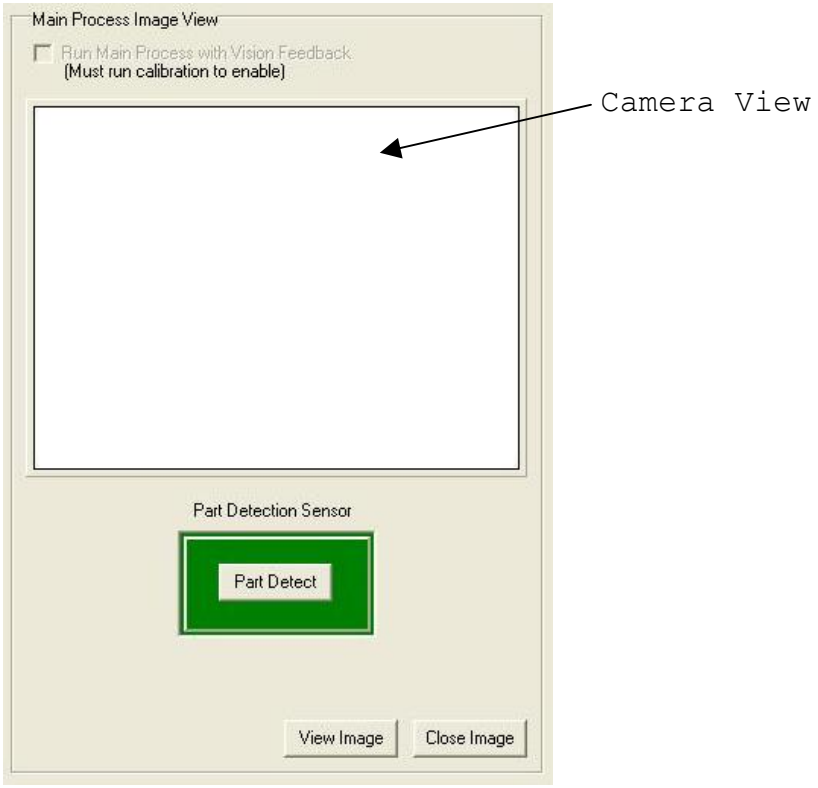


Figure 5-4 Main Process Image View

a. Run Main Process With Vision Feedback: After a successful calibration, when this check box is selected the vision feedback motion will be enabled when the "Run" button is pressed and a part is detected by the sensor.
b. View Image: The "View Image" button will display live video in the "Camera View" area.
c. Close Image: The "Close Image" button will shutdown the webcam.

d. Camera View: The camera view area will show the live video once the "View Image" button is pressed. If "Run Main Process with Vision Feedback" is checked and the main process is executed by pressing the "Run" button, a still image will appear after a part is detected by the "Part Detect Sensor".

4. Motion No.: "Motion No." displays the current move number when programming and running a motion profile.

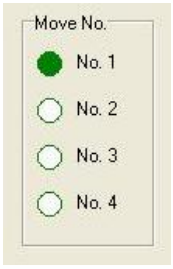


Figure 5-5 Move No.

5. Repeat Motion: "Repeat Motion" check boxes will enable the operator to repeat the corresponding axis motion profile until cancelled.

Figure 5-6 Repeat Motion

6. Motion Parameters

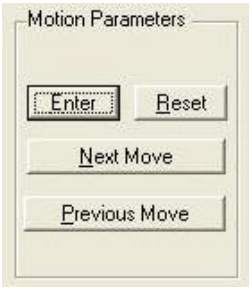


Figure 5-7 Motion Parameters

a. Enter: The "Enter" button saves the current move number (shown under "Move No.") motion profile parameters.
b. Reset: The "Reset" button erases the complete motion profile for all move numbers.
c. Next Move: Pressing the "Next Move" button will show the next move number's motion profile parameters for all the axes. Up to four moves can be programmed at one time.
d. Previous Move: Pressing the "Previous Move" button will show the previous move number's motion profile parameters for all the axes.

7. Run: Pressing the "Run" button will execute the current motion profile. The vision motion profile will run if the "Run Main Process with Vision Feedback" is checked.

8. Stop: Pressing the "Stop" button after the "Run" button will stop all axes from moving.
9. Home: Pressing the "Home" button will cause all axes to seek to the home position.

5.2.4.2 Calibration

From the Main screen, the Calibration screen is accessed by navigating to "Tools>Calibrate". This section defines the functions of the Calibration Menu.

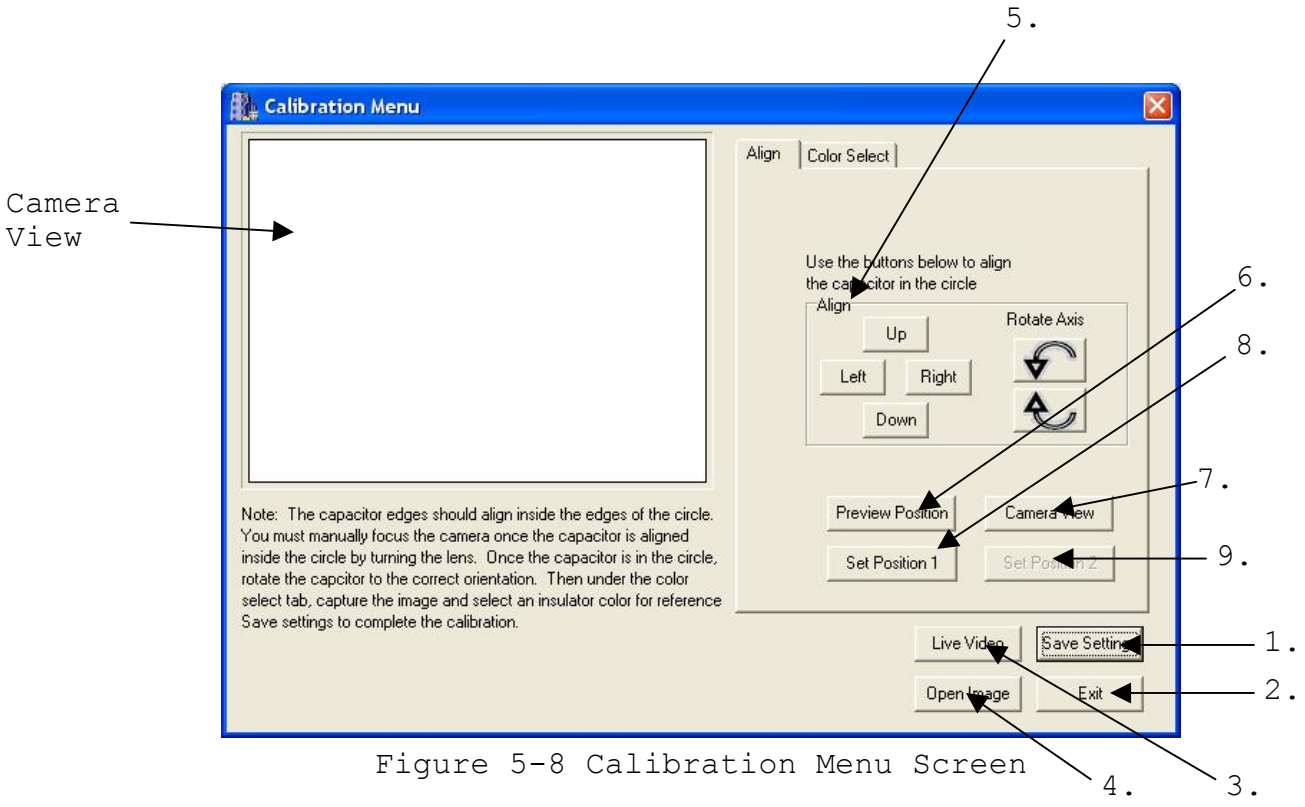


Figure 5-8 Calibration Menu Screen

The “Align” tab allows the operator to calibrate the capacitor position between the camera’s FOV center point and the gripper’s center point.

1. Save Settings: The “Save Settings” button will save the calibration settings.
2. Exit: Pressing the “Exit” button will exit the Calibration Menu.
3. Live Video: Pressing the “Live Video” button will enable the webcam and show the live video in the camera view area.
4. Open Image: Pressing the “Open Image” button will enable the operator to open an existing image.
5. Align

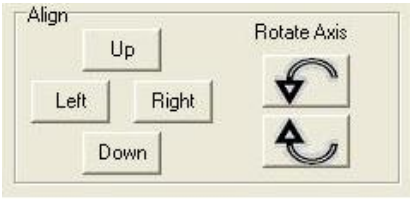


Figure 5-9 Align

a. Up: Pressing the “UP” button will jog the z-axis up at a velocity of 2 cm/s
b. Down: Pressing the “Down” button will jog the z-axis down at a velocity of 1.5 cm/s.

c. Left: Pressing the “Left” button will jog the x-axis to the left at a velocity of 1.5 cm/s.
d. Right: Pressing the “Right” button will jog the x-axis to the right at a velocity of 1.5 cm/s.
e. Rotate CCW: Pressing the “Rotate CCW” button will jog the theta axis counter clockwise at a velocity of 5 rev/s.
f. Rotate CW: Pressing the “Rotate CW” button will jog the theta axis clockwise at a velocity of 5 rev/s.

6. Preview Position: Pressing the “Preview Position” button will capture the current image and overlay a circle to see if the capacitor is aligned inside.
7. Camera View: Pressing the “Camera View” button will show the live video after the “Preview Position” button was pressed.
8. Set Position 1: Press “Set Position 1” to save the position when the center of the gripper is aligned with the center of the capacitor.
9. Set Position 2: Press “Set Position 2” to save the position when the center of the capacitor is aligned in the center of the circle shown when the “Preview Position” button is pressed.

The Calibration Menu's "Color Select" tab enables the operator to select the color insulator and set the orientation of the selected color insulator by physically positioning the desired color insulator on the capacitor in the correct orientation.

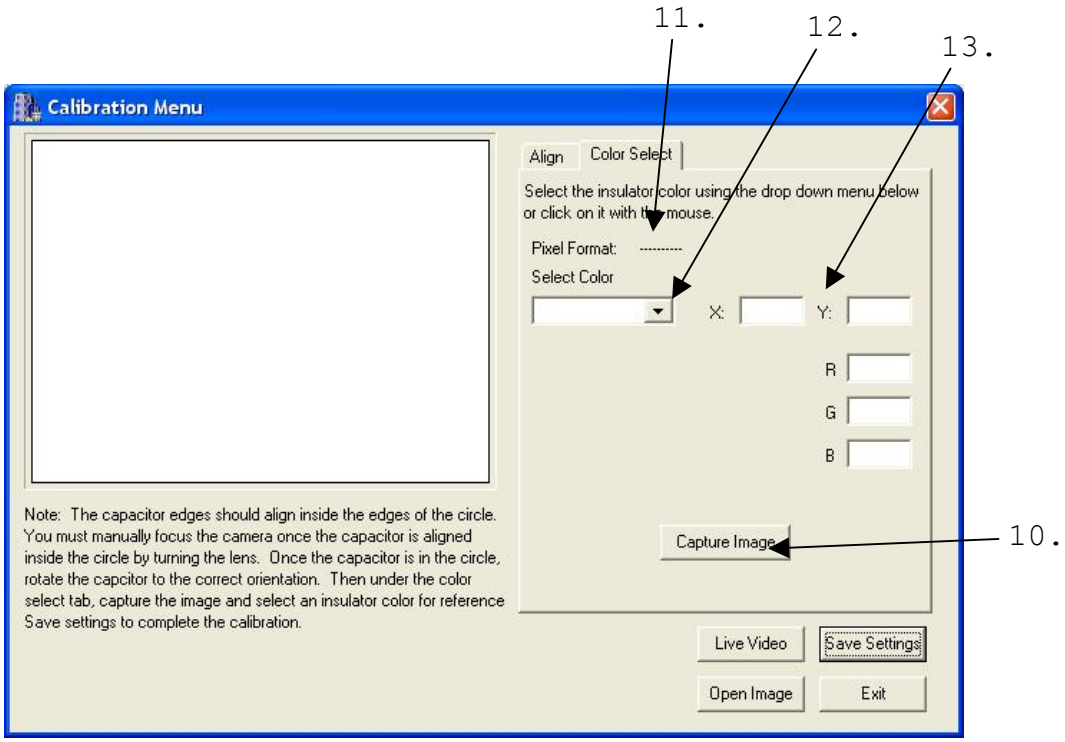


Figure 5-10 Calibration Menu Color Select

10. Capture Image: Pressing the "Capture Image" button will cause the webcam to take a snap shot of the current video and display it in the Camera View area.

11. Pixel Format: After the "Capture Image" button is pressed, the pixel bit format of the current graphic will be displayed here.

a. 6 – 32 Bit pixel format

b. 5 – 24 Bit pixel format

c. 4 – 16 Bit pixel format

d. 3 – 15 Bit pixel format

e. 2 – 8 Bit pixel format

f. 1 – 4 Bit pixel format

g. 0 – 1 Bit pixel format

12. Color Select Combo box: The “Color Select” combo box allows the operator to select the desired color insulator to position the capacitor to. The choices are blue, black, white, and green. After a color is selected, the image will have a red cross overlaid on the center of the selected insulator’s terminal.

13. Pixel Information: The Pixel Information section will display a pixel’s position and RGB byte value if the pixel is selected by clicking on it with the mouse.

5.2.4.3 Axis Status

The Axis Status screen displays the each axis velocity and position. The screen is automatically displayed after the “Run” button is pressed.

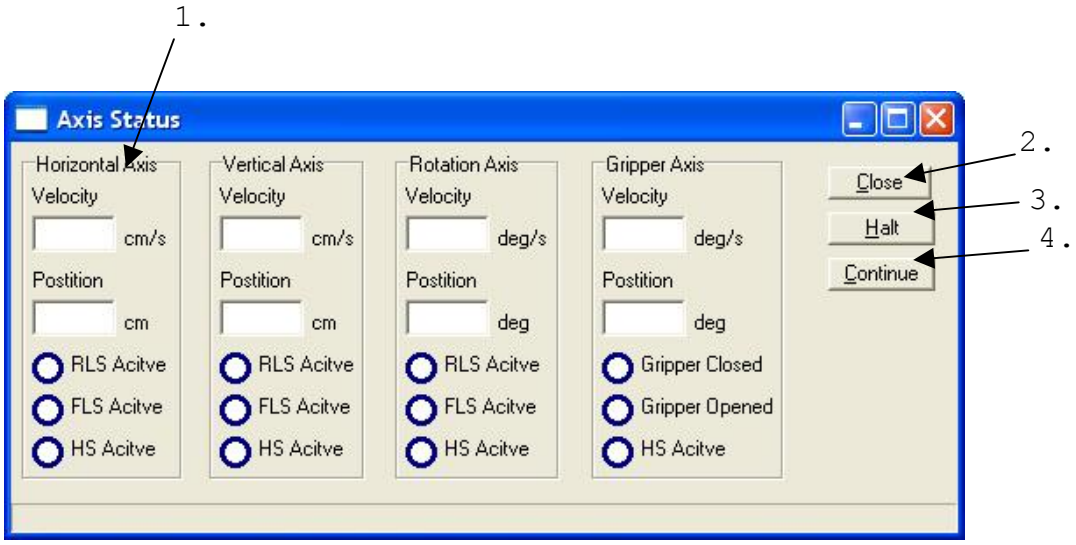


Figure 5-11 Axis Status Screen

1. Each axis has a velocity, position, limit switch state, and home switch state. The gripper axis shows and addition jaw position state.
2. Close: The "Close" button will exit the Axis Status screen.
3. Halt: The "Halt" button will halt all the axes and motion will come to a stop.
4. Continue: The "Continue" button will resume motion after the "Halt" button is pressed.

6 CONCLUSION

An autonomous robotic automation system with vision feedback was developed, based from a real world workforce problem, in an attempt to increase the speed and efficiency of production. There were many challenges, such as the use of off the shelf components to integrate and develop an HMI control and motion system with enough flexibility and growth capability to meet technology demands and to grow with the changing needs of the customer.

The system developed would be considered a prototype tool and not a production tool. There are many improvements necessary before a production tool could be released. The following section summarizes the manuscript followed by an improvement section to conclude this manuscript.

6.1 Summary

Chapter 1 was an introduction to automation history and the role automation plays in the manufacturing industry. In chapter 2, Robot Design, an introduction of typical robotic manipulator

configurations were presented along with a discussion of the cylindrical manipulator used in this project. Following the introduction, the kinematical relations of the cylindrical manipulator were developed using the Denavit-Hartenberg method. After the development of the kinematics, the velocity relationship between the joint velocities and the linear and angular velocities evolved by developing the Jacobian matrices relating each joint's CG to the base frame. Once the Jacobian matrices were developed, the dynamics of the cylindrical manipulator were created using the Euler-Lagrange equations. Concluding the chapter, the dynamic equations created a model, which was simulated in Matlab with a linearization feedback PD control design used to demonstrate a common control strategy.

Chapter 3 started with an introduction to digital image processing. Following the introduction, a discussion of color image processing using a personal computer was presented followed by the design and development of the image processing algorithm used to process the capacitor orientation. The chapter concluded with a presentation of the algorithm.

Chapter 4 presented the process and summarized the actual operation of the automated subroutines for this autonomous robotic automation system. Each process was summarized with

step by step instructions detailing the "behind the scenes" operations.

Chapter 5 described the hardware and software tools used to develop the autonomous robotic automation system with vision feedback. Two sections, Hardware and Software were broken out to detail and describe the functions of the hardware and software components used to develop this system.

6.2 Improvements

The following list depicts some of the system changes necessary to prepare a production tool to meet today's automation requirements to improve manufacturing throughput.

- Preloaded ball screw to prevent system backlash.
- Add a linear encoder near process area for position information to compensate for mechanical stresses.
- Replace gripper motor with a motor that has a minimum velocity of 5 rev/sec and a minimum resolution line count of 1000 with an accuracy of +/-1000 arc/sec.
- Redesign camera mount to view object closer to improve image resolution.
- Reconfigure manipulator arrangement so vertical axis is joint 2 and the horizontal axis is joint 3. This will eliminate link 3 attached to the gripper and will also improve the moment of inertia for the rotation joint by aligning the CG of the vertical axis with the CG of the rotation axis.

APPENDIX A: MATLAB SIMULATION CODE

Matlab Simulation Parabolic Blending Function

(parabolic_blending.m):

```
function [sys, x0] = parabolic_blending(t, x, u, flag);
num_of_var = 3;
T_0 = 0;
T_f = 5;
q0 = [-5; -10; -15;]; %zeros(num_of_var, 1);
qf = [80; 80; 80;];
a = 1.5 * (4 * (qf - q0) / T_f^2);
T_b = T_f/2 - sqrt(a.^2 .* T_f^2 - 4.*a.*(qf-q0)) ./ (2.*a);
v = a .* T_b;
q1=q0+a./2.*(T_b.^2);
q2=q1+(T_f.*v-2.*T_b.*v);

if (abs(flag) == 1)
%State Derivatives ( flag = 1 or -1 )
    sys = [];

elseif flag == 3

    %  Output Equations
    if(T_0 <= t & t <= T_b)
         q  = q0 + a./2 .* t^2;
         dq = a.*t;
         ddq = a;
    elseif(T_b < t & t < T_f - T_b)
         q = q1+ v .* (t - T_b);
         dq = v;
        ddq = a-a;
    elseif (t<=T_f)
        q = q0 + a./2 .* (T_b.^2) + v .* (T_f - 2*T_b) + v .* (t
- T_f + T_b) - a./2 .* ((t - T_f + T_b).^2);
       dq = v-a.*(t-T_f+T_b);
       ddq = -a;

    else
        q=qf;
        dq=qf-qf;
        ddq=a-a;
    end

    sys = [  q;
            dq;
           ddq];
```

```
elseif flag == 0
    %  Initial Conditions
  %  a = 1.5 * (4 * (qf - q0) / T_f^2);
   %  T_b  =  T_f/2  -  sqrt(a.^2  .*  T_f^2  -  4.*a.*(qf-q0))  ./
(2.*a);
   % v = a .* T_b;
    % sys=[state, 0, output,     input, 0, 0]
     sys = [0,     0, 3*num_of_var, 0, 0, 0];
    x0 = [];

end
```

Matlab Simulation Gain Function (GAIN.M):

```
%Evaluate Kv and Kp and multiply by e and edot, the error
tracking signals.
%Calculate u, input control.

function  [sys, x0]  = GAIN(t, x, u, flag);

joint_var = 3;          %Number of joint variables
dmp_ratio = 1;          %Critically damped system.  Recommended
for robotics
set_time = 0.5;           %1s settling time

Kd = 8/set_time;        %Derivative gain (velocity)
Kp = Kd^2/4;            %Proportional gain (position)

if (abs(flag) == 1)     %Return state derivatives
    sys = [];

elseif flag == 3        %Return output Equations
      e = u(1 : joint_var);
      edot = u(joint_var+1: 2*joint_var);
      sys = Kd * edot + Kp * e;

elseif flag == 0        %Assign initial Conditions
      sys = [0,     0, joint_var, 2*joint_var, 0, 0];     %sys =
[state, 0, output,     input, 0, 0]
      x0 = [];
end
```

Matlab Simulation M(q) Function (M_Q2.m)

```
%Evaluate M(q) and mulitply by u

function  [sys, x0]  = M_Q2(t, x, u, flag);

joint_var = 3;          %Number of joint variables
g = 9.81;               %Gravity (m/s^2)
m1 = 5.3;               %Mass of link 1 (kg)
m2 = 4.8;               %Mass of link 2 (kg)
m3 = 2.0;               %Mass of link 3 (kg)
Izz1 = 0.1079;          %Moment of inertia about the z axis of
link 1 (kgm^2)
Izz3 = 0.0073           %Moment of inertia about the z axis of
link 3 (kgm^2)

if (abs(flag) == 1)     %Return state derivatives
    sys = [];

elseif flag == 3        %Return output equation M(q) * u.
        q = u(1 : joint_var);
        qdot = u(joint_var+1 : 2*joint_var);
        input = u(2*joint_var+1 : 3*joint_var);
        theta = q(1);
        d2 = q(2);
        d3 = q(3);

        M = [0.203+6.8*d2^2 0.406 0.007; 0.813 14.6 0; 0.015 0
4.015];


        sys = M*input;

elseif flag == 0        %Assign initial Conditions
      sys = [0,     0, joint_var, 3*joint_var, 0, 0];     %sys =
[state, 0, output,     input, 0, 0]
      x0 = [];

end
```

Matlab Simulation N(q) Function (N_Q2.m):

```
%Calculate N(q)

function  [sys, x0]  = N_Q2(t, x, u, flag);

joint_var = 3;          %Number of joint variables
g = 9.81;               %Gravity (m/s^2)
m1 = 5.3;               %Mass of link 1 (kg)
m2 = 4.8;               %Mass of link 2 (kg)
m3 = 2.0;               %Mass of link 3 (kg)
Izz1 = 0.1079;          %Moment of inertia about the z axis of
link 1 (kgm^2)
Izz3 = 0.0073           %Moment of inertia about the z axis of
link 3 (kgm^2)

if (abs(flag) == 1)     %Return state derivatives
    sys = [];

elseif flag == 3        %Return output matrix N(q).  Will be a
3x1
        q = u(1 : joint_var);
        qdot = u(joint_var+1 : 2*joint_var);
        theta = q(1);
        d2 = q(2);
        d3 = q(3);
        theta_dot=qdot(1);
        d2_dot=qdot(2);
        d3_dot=qdot(3);

        N = [(13.6*d2*d2_dot*theta_dot)+(d2_dot*d3_dot);-
6.8*d2*theta_dot^2;19.6];

        sys = N;

elseif flag == 0        %Assign initial Conditions
      sys = [0,     0, joint_var, 2*joint_var, 0, 0];     %sys =
[state, 0, output,     input, 0, 0]
      x0 = [];

end
```

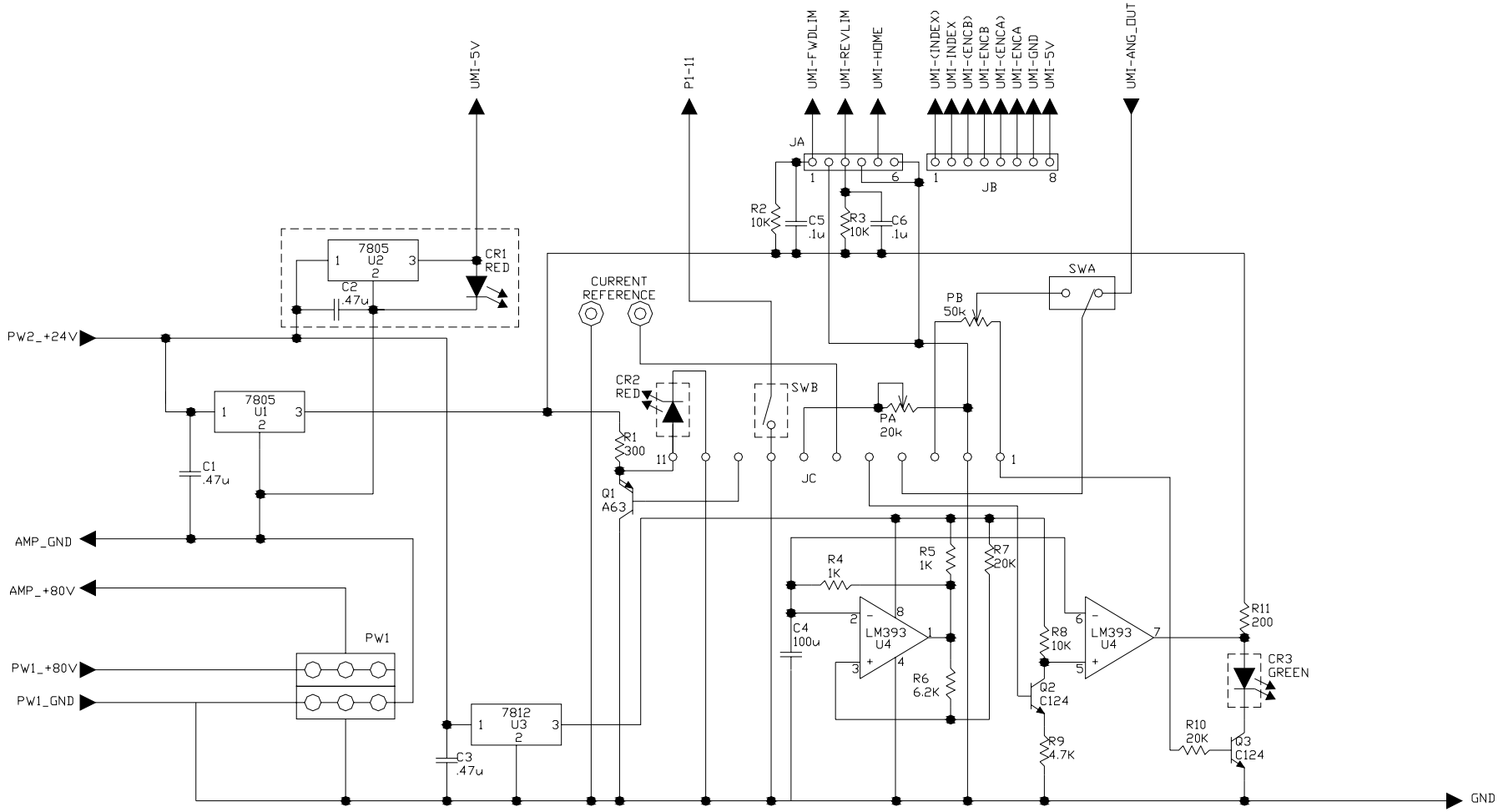
Matlab Simulation Dynamics Function (DYNAMICS2.m):

```
% hEvaluate M(q) given vector of angles (for trig. function) and
% Multiply the resulting matrix M(q) with an input vector.

function  [sys, x0]  = DYNAMICS2(t, x, tau, flag);

joint_var = 3;          %Number of joint variables
g = 9.81;               %Gravity (m/s^2)
m1 = 5.3;               %Mass of link 1 (kg)
m2 = 4.8;               %Mass of link 2 (kg)
m3 = 2.0;               %Mass of link 3 (kg)
Izz1 = 0.1079;          %Moment of inertia about the z axis of
link 1 (kgm^2)
Izz3 = 0.0073           %Moment of inertia about the z axis of
link 3 (kgm^2)

if (abs(flag) == 1)     %Return state derivatives

    q = x(1 : joint_var);
    qdot = x(joint_var+1 : 2*joint_var);
    theta = q(1);
    d2 = q(2);
    d3 = q(3);
    theta_dot = qdot(1);
    d2_dot = qdot(2);
    d3_dot = qdot(3);

    M = [0.203+6.8*d2^2 0.406 0.007; 0.813 14.6 0; 0.015 0
4.015];
    N = [(13.6*d2*d2_dot*theta_dot)+(d2_dot*d3_dot);-
6.8*d2*theta_dot^2;19.6];


    qdot2 = inv(M) * (tau - N);

    sys = [qdot; qdot2];

elseif flag == 3        %Return output vector
    q = x(1 : joint_var);
    qdot = x(joint_var+1 : 2*joint_var);
    theta = q(1);
    d2 = q(2);
    d3 = q(3);
    theta_dot = qdot(1);
    d2_dot = qdot(2);
```

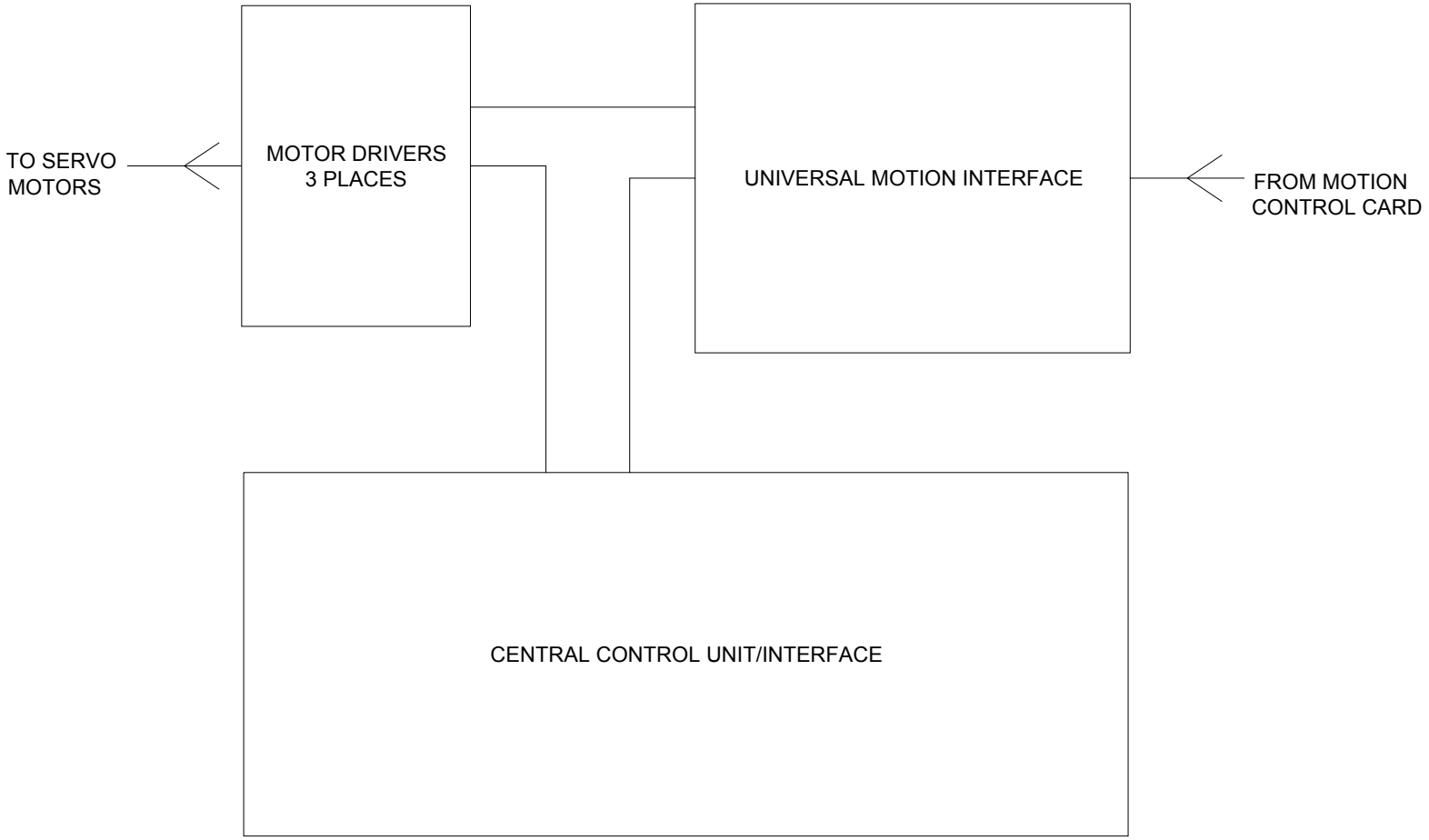
```
    d3_dot = qdot(3);

    M = [0.203+6.8*d2^2 0.406 0.007; 0.813 14.6 0; 0.015 0
4.015];
    N = [(13.6*d2*d2_dot*theta_dot)+(d2_dot*d3_dot);-
6.8*d2*theta_dot^2;19.6];

    qdot2 = inv(M) * (tau - N);

    sys = [q; qdot; qdot2];

elseif flag == 0        %Assign initial Conditions
    sys = [2*joint_var,     0, 3*joint_var, joint_var, 0, 0];
    x0 = zeros(1, 2*joint_var);

end
```

APPENDIX B: SCHEMATICS

Main Axis Interface Circuit:



System Interconnect Diagram:



APPENDIX C: HMI SOURCE CODE

Main Program (inmotion_proj.cpp)

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
USERES("inmotion_proj.res");
USEFORM("inmotion.cpp", motion);
USELIB("FlexBC32.lib");
USEFORM("Init_board.cpp", InitBoard);
USEFORM("Axis_status.cpp", AxisStatus);
USEUNIT("Move.cpp");
USEFORM("Calibration.cpp", CalibrateMenu);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
        try
        {
                Application->Initialize();
                Application->CreateForm(__classid(Tmotion), &motion);
                Application->CreateForm(__classid(TInitBoard), &InitBoard);
                Application->CreateForm(__classid(TAxisStatus), &AxisStatus);
                Application->CreateForm(__classid(TCalibrateMenu), &CalibrateMenu);
                Application->Run();
        }
        catch (Exception &exception)
        {
                Application->ShowException(&exception);
        }
        return 0;
}
//---------------------------------------------------------------------------
```

Main Diaglog Motion Form Code (immotion.cpp)

```
//---------------------------------------------------------------------------

#include <vcl.h>
#include <io.h>
#pragma hdrstop
#include "inmotion.h"
#include "Init_board.h"
#include "Axis_status.h"
#include "Calibration.h"
#include "axesparam.h"
#include "Move.h"
#include <stdio.h>
#include <stdlib.h>
#include <dstring.h>
#include <process.h>
#include <math.h>
#define HOST 0xFF
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "VIDEOOCXLib_OCX"
#pragma link "IEVect"
#pragma link "ieview"
#pragma link "ImageEnView"
#pragma link "VideoCap"
#pragma resource "*.dfm"
//---------------------------------------------------------------------------
//Function Prototypes written in inmotion.cpp
//View velocity and postion on the screen.
void DisplayAxesStatus(u16 ASHorz, u16 ASRot, u16 ASVert);
//Errorhandler will print the error discription to screen
```

```
void ErrorHandler(i32 errorCode, u16 commandID, u16 resourceID);
//Function to load the parameters to each axis
void LoadParameters(u8 mAxis, f64 mAcc, f64 mVel, i32 mPos);
//------------------------------------------------------------------------
//Function prototypes written in calibration.cpp
//Function to get raw data from image and store in containers.
void GetRawData(Graphics::TBitmap *Image);
//Function to find and check for ccrrect color
bool ColorFind(int LowerBnd[3], int UpperBnd[3], int Color);

//------------------------------------------------------------------------
//Local Variables defined using NI's assigned types
u8 boardID=1;               //Board Identification number
u8 next=0;                      //Next point position to specify
u8 axis=1;              //Axis Number
u8 primaryFBK=0x21;             //Encoder number.  Set to Encoder 1.
u8 dac=0x31;                    //DAC output.  Set to DAC 1.
u8 enableAxes=0;                //Bitmap for enabling axes 1 through 4.
u8 limitPolarity=0x14;          //Set polarity for limit switches. Active High.
u8 limitEnable=0x0A;            //Enabling limit switches. Axes 1, 3, and 4.
u8 BPon=0x10;                   //Turning on gripper open command
u8 BPoff=0x0E;                  //Tuning off axes 1-3 motion led's
u16 home=0x001A;                //Enabling home switches. Axes 1, 3, and 4.
u8 FLS;                         //Return bitmap for forward limit status.
u8 RLS;                         //Return bitmap for reverse limit status.
u16 homePolarity=0x001E;        //Setting home polarity. Active low.
u16 moveCriteria=0x0019;        //Move complete criteria.
u16 Axes1_4=0;                  //Group of Axes to Move, 1 - 4. Setting so no axes move
u16 followingError=32767;       //Maximum following error to allow on closed loop axes.
u16 findHome1=0x0002;           //Setting the find home direction and final approach for
axis 1.
```

```
u16 findHome3=0x0005;           //Setting the find home direction and final approach for
axis 3.
u16 findHome4;                  //Setting the find home direction and final approach for
axis 4.
u16 findIndex=0;                //Search for index in the forward direction.
u16 AxisRtn1, AxisRtn2, AxisRtn3, AxisRtn4;       //Axis status variable for each axis.
u16 csr;            //Communication Status Register
u16 TriggerInputs;              //Trigger inputs to read gripper position.
i16 indexOffset=0;              //Set offset to zero.
i16 FollowErrorRtn;             //Variable used by NI function to return following error.
i32 status=0;               //Return Status of functions
i32 positionRet[4];             //An array used to store each axis current position.1-4.
i32 velocityRet[4];             //An array used to store each axis current velocity.1-4.
i32 RotInitialPos;              //Last postion stored for rotation axis.
i32 targetPos;                 //Target Position in counts
f64 accelDecel;                 //Acceleration value in rad/s^2
f64 velocity;            //Velocity value in RPM
u32 scanVar;             //scanf wants a u16 to put numeric values in.
int AngleDif;                   //Angle to rotate cap to desired location.
double Angle1;                  //Angle from desired location
double Angle2;                  //Angle from current location
BYTE MotionImageData[222][74];  //Image raw data storage container

//Checking variables used to make sure certain sequences and buttons are followed.
bool Enter = false;             //Variable to check and see if Enter button has been
pressed.
bool InitCheck = false;         //Variable to check and see if board has been
initialized.
bool ParamEntered = false;      /*Variable to make sure parameters have been entered.
                                Used in the "NextbutClick" definition.*/
bool RunButton = false;         //Make sure run button has been pressed.
```

```
//Variables for position entered check.
bool HorzParamPos[4] = {false, false, false, false};
bool VertParamPos[4] = {false, false, false, false};
bool RotParamPos[4] = {false, false, false, false};
bool GripParamPos[4] = {false, false, false, false};
bool ResetCheck = false;        //Variable to check and see if reset button has been
pressed.
bool OpenCheck = false;         //Bool variable to check and see if a saved motion
profile has been opened.
bool CamFuncCheck = false;      //Bool to check if camera videoOCX functions are working
bool GripperOpened;              //Gripper position return variable. (Open)
bool GripperClosed;              //Gripper position return variable. (Closed)
bool EnCapMotion = false;        //Enable the webcam to control cap rotation.
bool CameraOnCheck = false;          //Check to see if camera is on before exiting
bool ColorFound;              //Bool to see if color was found correctly

//Repeat motion checkbox variables.
bool RepeatHorz = false;
bool RepeatVert = false;
bool RepeatRot = false;
bool RepeatGrip = false;

int gripoc[4];                  //Variable to store gripper open/close state

//Variables for modal error handling
u16 commandID;           //The commandID of the function
u16 resource;            //The resource ID
i32 errorCode;                 //The error generated

//Variable to capture raw image data
long m_ImageHandle;
```

```
//Variable defined in axis_status.cpp
extern HaltCheck;
//Variables declared in calibration.cpp
extern PixelxDes;                //Variable to hold middle of connector y
extern PixelyDes;                //Variable to hold middle of connecto x
extern Pixelx;                  //Pixel value saved to store desired postion along x
extern Pixely;                  //Pixel value saved to store desired postion along y
extern ColorLowerBound[3];      //Array to hold lower threshold for RGB (R=0, G=1, B=2);
extern ColorUpperBound[3];      //Array to hold upper threshold for RGB (R=0, G=1, B=2);
extern ColorSelected;           //Color selected from calibration menu.
extern CalMenuCheck;            //Check variable for calibration menu type bool.

PID PIDvalues;                  //The PID class created by NI.

TRect ColorCap;                 //Rectangular class created by Borland

//Creating an array of pointers to the classes of axes paramaters and  motion.
HorizontalAxis *HorzArray[4];
VerticalAxis *VertArray[4];
RotationAxis *RotArray[4];
GripperAxis *GripperArray[4];

//Defining pointers to classes
Tmotion *motion;
//Dynamically allocating memory from the heap
MoveThread *MoveComplete = new MoveThread(true);

Graphics::TBitmap *MotionPicture=new Graphics::TBitmap();
```

```
//---------------------------------------------------------------------------

__fastcall Tmotion::Tmotion(TComponent* Owner)
        : TForm(Owner)
{
        u8 trig;

        //Initialize the gripper open/close array to zero.
        for(int i=0;i<4 ;i++){
                gripoc[i]=-1;
        }
         //Initializing the return arrays to zero
        for(int i=0; i<4; i++){
                positionRet[i]=0;
                velocityRet[i]=0;
        }

        //Defining some intial parameters.
        accelDecel=50;
        velocity=200;
        targetPos=0;
        PIDvalues.ilim=1000;
        PIDvalues.td=2;
        PIDvalues.aff=0;
        PIDvalues.vff=0;
        PIDvalues.kp=5;
        PIDvalues.ki=3;
        PIDvalues.kd=1;

        //Coloring the motion number circles to the correct color.
```

```
        MoveNo1->Brush->Color=clGreen;
        MoveNo2->Brush->Color=clWhite;
        MoveNo3->Brush->Color=clWhite;
        MoveNo4->Brush->Color=clWhite;

        //Dynamically allocating memory for my axis parameter storage.
        for(int i=0;i<4;i++){
        HorzArray[i] = new HorizontalAxis();
        VertArray[i] = new VerticalAxis();
        RotArray[i] = new RotationAxis();
        GripperArray[i] = new GripperAxis();
        }

        //Reseting position to zero.
        for(axis=1;axis<5;axis++){
                if(status == NIMC_noError){
                status = flex_reset_pos(boardID,axis,0,0,HOST);
                }
        }

        //Setting breakpoint outputs off for axis 1-3 and on for axis 4
        if(status == NIMC_noError){
                status = flex_set_breakpoint_momo(boardID, 0, BPon, BPoff, HOST);
        }

        //Setting the gripper position inputs to triggers 1 and 2.
        for(trig=1; trig<3; trig++){
                if(status == NIMC_noError){
                        status = flex_configure_hs_capture(boardID, trig,
NIMC_HS_NON_INVERTING_DI, 0);
                }
```

```
}

/*Disabling and Graying out the edit input boxes on the gui until enable
axes is checked to show them*/
VelHorz->Enabled = false;
VelHorz->Color = clBtnFace;
AccHorz->Enabled = false;
AccHorz->Color = clBtnFace;
PosHorz->Enabled = false;
PosHorz->Color = clBtnFace;

VelVert->Enabled = false;
VelVert->Color = clBtnFace;
AccVert->Enabled = false;
AccVert->Color = clBtnFace;
PosVert->Enabled = false;
PosVert->Color = clBtnFace;

VelRot->Enabled = false;
VelRot->Color = clBtnFace;
AccRot->Enabled = false;
AccRot->Color = clBtnFace;
PosRot->Enabled = false;
PosRot->Color = clBtnFace;

VelGrip->Enabled = false;
VelGrip->Color = clBtnFace;
AccGrip->Enabled = false;
AccGrip->Color = clBtnFace;
PosGrip->Enabled = false;
PosGrip->Color = clBtnFace;
```

```
        GripOC->Enabled = false;
        GripCloseBtn->Enabled = false;
        GripOpenBtn->Enabled = false;

        //Setting the video panel to visible
        VideoPanel->Visible = true;

        //Defining color circle rectangle
        ColorCap.Left=40; ColorCap.Right=280;
        ColorCap.Top=0;  ColorCap.Bottom=240;

}
//---------------------------------------------------------------------------
//Function call to open a file and place saved profile into edit boxes
void __fastcall Tmotion::Open1Click(TObject *Sender)
{
        if(ResetCheck){
                OpenCheck = true;
                int line = 0;
                OpenDialog1->Filter = "Text files (*.txt)|*.TXT";
                //Opening the file and putting the variables line by line into the rich
text edit form.
                SaveDoc->Lines->Clear();
                if(OpenDialog1->Execute()){
                        SaveDoc->Lines->LoadFromFile(OpenDialog1->FileName);
                }
                //Passing variables from rich edit box into my axes variables.
                for(int i=0; i<4; i++){
```

```
                        HorzArray[i]->VelHorzInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        HorzArray[i]->AccHorzInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        HorzArray[i]->PosHorzInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        HorzArray[i]->OrderHorz=StrToInt(SaveDoc->Lines->Strings[line]);
                        line++;
                        RotArray[i]->VelRotInt=StrToFloat(SaveDoc->Lines->Strings[line]);
                        line++;
                        RotArray[i]->AccRotInt=StrToFloat(SaveDoc->Lines->Strings[line]);
                        line++;
                        RotArray[i]->PosRotInt=StrToFloat(SaveDoc->Lines->Strings[line]);
                        line++;
                        RotArray[i]->OrderRot=StrToInt(SaveDoc->Lines->Strings[line]);
                        line++;
                        VertArray[i]->VelVertInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        VertArray[i]->AccVertInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        VertArray[i]->PosVertInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        VertArray[i]->OrderVert=StrToInt(SaveDoc->Lines->Strings[line]);
                        line++;
```

```
                        GripperArray[i]->VelGripInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        GripperArray[i]->AccGripInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        GripperArray[i]->PosGripInt=StrToFloat(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        GripperArray[i]->OrderGrip=StrToInt(SaveDoc->Lines-
>Strings[line]);
                        line++;
                        gripoc[i]=StrToFloat(SaveDoc->Lines->Strings[line]);
                        line++;
                }

                if(HorzArray[0]->VelHorzInt && HorzArray[0]->AccHorzInt){
                        VelHorz->Text=HorzArray[0]->VelHorzInt;
                        AccHorz->Text=HorzArray[0]->AccHorzInt;
                        PosHorz->Text=HorzArray[0]->PosHorzInt;
                        HorzOrdRG->ItemIndex=HorzArray[0]->OrderHorz;
                        HorzParamPos[0]=true;
                        ParamEntered=true;
                }
                if(RotArray[0]->VelRotInt && RotArray[0]->AccRotInt){
                        VelRot->Text=RotArray[0]->VelRotInt;
                        AccRot->Text=RotArray[0]->AccRotInt;
                        PosRot->Text=RotArray[0]->PosRotInt;
                        RotOrdRG->ItemIndex=RotArray[0]->OrderRot;
                        RotParamPos[0]=true;
                        ParamEntered=true;
```

```
}
if(VertArray[0]->VelVertInt && VertArray[0]->AccVertInt){
        VelVert->Text=VertArray[0]->VelVertInt;
        AccVert->Text=VertArray[0]->AccVertInt;
        PosVert->Text=VertArray[0]->PosVertInt;
        VertOrdRG->ItemIndex=VertArray[0]->OrderVert;
        VertParamPos[0]=true;
        ParamEntered=true;
}
if(GripperArray[0]->VelGripInt && GripperArray[0]->AccGripInt){
        VelGrip->Text=GripperArray[0]->VelGripInt;
        AccGrip->Text=GripperArray[0]->AccGripInt;
        PosGrip->Text=GripperArray[0]->PosGripInt;
        GripOrdRG->ItemIndex=GripperArray[0]->OrderGrip;
        GripParamPos[0]=true;
        ParamEntered=true;
}

GripOC->ItemIndex=gripoc[0];

for(int i=1; i<4; i++){
        if(HorzArray[i]->VelHorzInt && HorzArray[i]->AccHorzInt){
                HorzParamPos[i]=true;
        }
        if(RotArray[i]->VelRotInt && RotArray[i]->AccRotInt){
                RotParamPos[i]=true;
        }
        if(VertArray[i]->VelVertInt && VertArray[i]->AccVertInt){
                VertParamPos[i]=true;
        }
        if(GripperArray[i]->VelGripInt && GripperArray[i]->AccGripInt){
```

```
                                GripParamPos[i]=true;
                        }
                }

        }
        else{
                Application->MessageBox("Press the Reset button before opening a new
motion profile.","Sequence Error",MB_OK);
        }

}
//---------------------------------------------------------------------------
//Function call to save current motion profile into a txt file.
void __fastcall Tmotion::SaveAs1Click(TObject *Sender)
{

        //Saving the variables to the richedit box and then putting the richedit box into
a file.
        //File will be saved as a .txt notepad file.
        SaveDoc->Lines->Clear();

        for(int i=0; i<4; i++){

                SaveDoc->Lines->Add(FloatToStr(HorzArray[i]->VelHorzInt));
                SaveDoc->Lines->Add(FloatToStr(HorzArray[i]->AccHorzInt));
                SaveDoc->Lines->Add(FloatToStr(HorzArray[i]->PosHorzInt));
                SaveDoc->Lines->Add(IntToStr(HorzArray[i]->OrderHorz));

                SaveDoc->Lines->Add(FloatToStr(RotArray[i]->VelRotInt));
                SaveDoc->Lines->Add(FloatToStr(RotArray[i]->AccRotInt));
                SaveDoc->Lines->Add(FloatToStr(RotArray[i]->PosRotInt));
```

```
                SaveDoc->Lines->Add(IntToStr(RotArray[i]->OrderRot));

                SaveDoc->Lines->Add(FloatToStr(VertArray[i]->VelVertInt));
                SaveDoc->Lines->Add(FloatToStr(VertArray[i]->AccVertInt));
                SaveDoc->Lines->Add(FloatToStr(VertArray[i]->PosVertInt));
                SaveDoc->Lines->Add(IntToStr(VertArray[i]->OrderVert));

                SaveDoc->Lines->Add(FloatToStr(GripperArray[i]->VelGripInt));
                SaveDoc->Lines->Add(FloatToStr(GripperArray[i]->AccGripInt));
                SaveDoc->Lines->Add(FloatToStr(GripperArray[i]->PosGripInt));
                SaveDoc->Lines->Add(IntToStr(GripperArray[i]->OrderGrip));

                SaveDoc->Lines->Add(FloatToStr(gripoc[i]));
        }

        if(SaveDialog1->Execute()){
                SaveDialog1->Filter = "Text files (*.txt)|*.TXT";
                SaveDoc->Lines->SaveToFile(SaveDialog1->FileName);
        }

}
//---------------------------------------------------------------------------
//Function call to exit program if EXIT BUTTON is pressed
void __fastcall Tmotion::Exit1Click(TObject *Sender)
{

        //Deleting arrays from memory
        for(int i=0;i<4;i++){
                delete HorzArray[i];
                delete VertArray[i];
                delete RotArray[i];
```

```
                delete GripperArray[i];
        }

        MoveComplete->Terminate();

        if(CameraOnCheck){
                //Stop video
                CamFuncCheck = MotionVideoOCX->Close();

                if(CamFuncCheck){
                        Application->MessageBox("Video capture device turned
off.","Message",MB_OK);
                }
                else{
                        Application->MessageBox("Video capture device failed to turn
off.","Error",MB_OK);
                }
        }

        delete MotionPicture;

        Close();
}
//---------------------------------------------------------------------------
//Function call to initialize board
void __fastcall Tmotion::InitializeBoard1Click(TObject *Sender)
{
        //Show initialize board form
        InitBoard->Visible=true;
        InitBoard->OkBut->Enabled = false;
        InitBoard->PBar1->Position=0;
```

```
}
//---------------------------------------------------------------------------
//Function call to reset trajectory
void __fastcall Tmotion::NewTrajectory1Click(TObject *Sender)
{
        ResbutClick(Sender);
}
//---------------------------------------------------------------------------
//Function call for ENTER BUTTON
void __fastcall Tmotion::Enter1Click(TObject *Sender)
{
        EnterbutClick(Sender);
}
//---------------------------------------------------------------------------
//Function call for RUN BUTTON
void __fastcall Tmotion::Run1MenuClick(TObject *Sender)
{
        RunButClick(Sender);
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::CamFormatClick(TObject *Sender)
{
        CamFuncCheck = ImageEnVideoView1->DoConfigureFormat();

        if( !CamFuncCheck ){

                MessageDlg("Configure Format dialog not available",mtInformation,
TMsgDlgButtons() << mbOK,0);
        }
}
```

```
//---------------------------------------------------------------------------

void __fastcall Tmotion::CamSettingsClick(TObject *Sender)
{
        CamFuncCheck = ImageEnVideoView1->DoConfigureSource();

        if( !CamFuncCheck ){

                MessageDlg("Configure Source dialog not available",mtInformation,
TMsgDlgButtons() << mbOK,0);
        }
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::Calibrate1Click(TObject *Sender)
{

        if(HorzEnable->Checked && RotEnable->Checked && VertEnable->Checked &&
GripEnable->Checked){
                CalibrateMenu->Visible=true;
        }
        else{
                Application->MessageBoxA("All axes must be enabled to run
calibration.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::EnVisionChkClick(TObject *Sender)
{
        EnCapMotion = true;
```

```
        if(EnVisionChk->Checked){
                EnCapMotion = false;
        }

}
//---------------------------------------------------------------------------

void __fastcall Tmotion::ViewbutClick(TObject *Sender)
{
        VideoPanel->Visible = true;
        ImagePanel->Visible = false;
        CameraOnCheck = true;

        MotionVideoOCX->SetResolution(320,240);

        //Initialize video camera
        CamFuncCheck = MotionVideoOCX->Init();

        //View live video
        if(CamFuncCheck){
                CamFuncCheck = MotionVideoOCX->SetPreview(true);

        }
        else{
                Application->MessageBoxA("Video Capture device is in use or has not
initialized.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------
void __fastcall Tmotion::CloseImageButClick(TObject *Sender)
```

```
{
        if(CameraOnCheck){
                //Stop video
                CamFuncCheck = MotionVideoOCX->Close();

                if(CamFuncCheck){
                        Application->MessageBox("Video capture device turned
off.","Message",MB_OK);
                }
                else{
                        Application->MessageBox("Video capture device failed to turn
off.","Error",MB_OK);
                }
        }
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::GripCloseBtnClick(TObject *Sender)
{
        //Checking to see if gripper is closed already
        if(status == NIMC_noError){
                flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
        }

        GripperClosed = 0x0002 & TriggerInputs;

         if(!GripperClosed){
                //Closing the gripper.
                if(status == NIMC_noError){
                        status = flex_set_breakpoint_momo(boardID, 0, 0x00, 0x10, HOST);
                }
```

```
         }else{
                Application->MessageBox("Gripper is already closed","Error",MB_OK);
         }

         AxisStatus->GripperClosedLt->Brush->Color=clRed;
         AxisStatus->GripperOpenLt->Brush->Color=clWhite;
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::GripOpenBtnClick(TObject *Sender)
{
        //Checking to see if gripper is opened already
        if(status == NIMC_noError){
                flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
        }

        GripperOpened = 0x0004 & TriggerInputs;

        if(!GripperOpened){
        //Opening the gripper.
                if(status == NIMC_noError){
                        status = flex_set_breakpoint_momo(boardID, 0, 0x10, 0x00, HOST);
                }
        }else{
                Application->MessageBox("Gripper is already open","Error",MB_OK);
         }

         AxisStatus->GripperClosedLt->Brush->Color=clWhite;
         AxisStatus->GripperOpenLt->Brush->Color=clRed;
}
//---------------------------------------------------------------------------
```

```
void __fastcall Tmotion::EnterbutClick(TObject *Sender)
{
        //Retrieving data from Tedit boxes and setting order of target positions.
        float VelLimit = 0;
        float AccLimit = 0;

        if(next<4){
                switch(next){
                        //Next button has not been pressed yet
                        case 0:
                        //Horizontal Axis
                        if(VelHorz->Text!="" && AccHorz->Text!="" && PosHorz->Text!=""){
                                VelLimit = StrToFloat(VelHorz->Text);
                                AccLimit = StrToFloat(AccHorz->Text);
                                //Check and see if vel and acc are with min and max
limits.
                                if((VelLimit<=15 && VelLimit>0)&&(AccLimit<=15 &&
AccLimit>0)){
                                        HorzArray[0]->VelHorzInt=StrToFloat(VelHorz-
>Text);
                                        HorzArray[0]->AccHorzInt=StrToFloat(AccHorz-
>Text);
                                        HorzArray[0]->PosHorzInt=StrToFloat(PosHorz-
>Text);
                                        //Making sure that an order has been selected.
                                        if(HorzOrdRG->ItemIndex!=-1){
                                                ParamEntered = true;
                                                HorzArray[0]->OrderHorz=HorzOrdRG-
>ItemIndex;
```

```
                                                //Making sure that the position has been
altered.
                                                if(PosHorz->Modified || VelHorz->Modified
|| AccHorz->Modified){
                                                        HorzParamPos[0] = true;

                                                }else{
                                                        if(RunButton){
                                                                HorzParamPos[0] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                HorzParamPos[0] = false;
                                                Application->MessageBox("Must select and
order for the horizontal axis","Error",MB_OK);
                                                break;
                                        }

                                        PosHorz->Modified = false;
                                }else{
                                        Application->MessageBox("Horizontal velocity and
acceleration value must be between 0 - 15","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Rotational Axis
                        if(VelRot->Text!="" && AccRot->Text!="" && PosRot->Text!=""){
                                VelLimit = StrToFloat(VelRot->Text);
                                AccLimit = StrToFloat(AccRot->Text);
```

```
                                //Check and see if vel and acc are with min and max
limits.
                                if((VelLimit<=20 && VelLimit>0)&&(AccLimit<=20 &&
AccLimit>0)){
                                        RotArray[0]->VelRotInt=StrToFloat(VelRot->Text);
                                        RotArray[0]->AccRotInt=StrToFloat(AccRot->Text);
                                        RotArray[0]->PosRotInt=StrToFloat(PosRot->Text);
                                        //Making sure that an order has been selected.
                                        if(RotOrdRG->ItemIndex!=-1){
                                                ParamEntered = true;
                                                RotArray[0]->OrderRot=RotOrdRG-
>ItemIndex;
                                                //Making sure that the position has been
altered.
                                                if(PosRot->Modified || VelRot->Modified
|| AccRot->Modified){
                                                        RotParamPos[0] = true;
                                                }else{
                                                        if(RunButton){
                                                                RotParamPos[0] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                RotParamPos[0] = false;
                                                Application->MessageBox("Must select and
order for the rotational axis","Error",MB_OK);
                                                break;
                                        }

                                        PosRot->Modified = false;
```

```
                                }else{
                                        Application->MessageBox("Rotational velocity and
acceleration value must be between 0 - 20","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Vertical Axis
                        if(VelVert->Text!="" && AccVert->Text!="" && PosVert->Text!=""){
                                VelLimit = StrToFloat(VelVert->Text);
                                AccLimit = StrToFloat(AccVert->Text);
                                //Check and see if vel and acc are with min and max
limits.
                                if((VelLimit<=15 && VelLimit>0)&&(AccLimit<=15 &&
AccLimit>0)){
                                        VertArray[0]->VelVertInt=StrToFloat(VelVert-
>Text);
                                        VertArray[0]->AccVertInt=StrToFloat(AccVert-
>Text);
                                        VertArray[0]->PosVertInt=StrToFloat(PosVert-
>Text);
                                        //Making sure that an order has been selected.
                                        if(VertOrdRG->ItemIndex!=-1){
                                                ParamEntered = true;
                                                VertArray[0]->OrderVert=VertOrdRG-
>ItemIndex;
                                                //Making sure that the position has been
altered.
                                                if(PosVert->Modified || VelVert->Modified
|| AccVert->Modified){
                                                        VertParamPos[0] = true;
```

```
                                                }else{
                                                        if(RunButton){
                                                                VertParamPos[0] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                VertParamPos[0] = false;
                                                Application->MessageBox("Must select and
order for the vertical axis","Error",MB_OK);
                                                break;
                                        }

                                        PosVert->Modified = false;
                                }else{
                                        Application->MessageBox("Vertical velocity and
acceleration value must be between 0 - 15","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Gripper Axis
                        if(VelGrip->Text!="" && AccGrip->Text!="" && PosGrip->Text!=""){
                                VelLimit = StrToFloat(VelGrip->Text);
                                AccLimit = StrToFloat(AccGrip->Text);
                                //Check and see if vel and acc are with min and max
limits.
                                if((VelLimit<=40 && VelLimit>0)&&(AccLimit<=80 &&
AccLimit>0)){
                                        GripperArray[0]->VelGripInt=StrToFloat(VelGrip-
>Text);
```

```
                                        GripperArray[0]->AccGripInt=StrToFloat(AccGrip-
>Text);
                                        GripperArray[0]->PosGripInt=StrToFloat(PosGrip-
>Text);
                                        //Making sure that an order has been selected.
                                        if(GripOrdRG->ItemIndex!=-1){
                                                ParamEntered = true;
                                                GripperArray[0]->OrderGrip=GripOrdRG-
>ItemIndex;
                                                //Making sure that the position has been
altered.
                                                if(PosGrip->Modified || VelGrip->Modified
|| AccGrip->Modified){
                                                        GripParamPos[0] = true;
                                                }else{
                                                        if(RunButton){
                                                                GripParamPos[0] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                GripParamPos[0] = false;
                                                Application->MessageBox("Must select and
order for the gripper axis","Error",MB_OK);
                                                break;
                                        }

                                        PosGrip->Modified = false;
                                }else{
                                        Application->MessageBox("Gripper velocity value
must be between 0 - 40 and acceleration between 0 - 80.","Value Error",MB_OK);
```

```
                                        break;
                                }
                        }
                        Enter = true;
                        gripoc[0] = GripOC->ItemIndex;
                        break;

                        //Next button has been pressed once
                        case 1:
                        //Horizontal Axis
                        if(VelHorz->Text!="" && AccHorz->Text!="" && PosHorz->Text!=""){
                                VelLimit = StrToFloat(VelHorz->Text);
                                AccLimit = StrToFloat(AccHorz->Text);
                                if((VelLimit<=15 && VelLimit>0)&&(AccLimit<=15 &&
AccLimit>0)){
                                        HorzArray[1]->VelHorzInt=StrToFloat(VelHorz-
>Text);
                                        HorzArray[1]->AccHorzInt=StrToFloat(AccHorz-
>Text);
                                        HorzArray[1]->PosHorzInt=StrToFloat(PosHorz-
>Text);
                                        //Making sure that an order has been selected.
                                        if(HorzOrdRG->ItemIndex!=-1){
                                                HorzArray[1]->OrderHorz=HorzOrdRG-
>ItemIndex;
                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosHorz->Modified || VelHorz->Modified
|| AccHorz->Modified){
                                                        HorzParamPos[1] = true;
```

```
                                                }else{
                                                        if(RunButton){
                                                                HorzParamPos[1] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                HorzParamPos[1] = false;
                                                Application->MessageBox("Must select and
order for the horizontal axis","Error",MB_OK);
                                                break;
                                        }

                                        PosHorz->Modified = false;
                                }else{
                                        Application->MessageBox("Horizontal velocity and
acceleration value must be between 0 - 15","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Rotational Axis
                        if(VelRot->Text!="" && AccRot->Text!="" && PosRot->Text!=""){
                                VelLimit = StrToFloat(VelRot->Text);
                                AccLimit = StrToFloat(AccRot->Text);
                                if((VelLimit<=20 && VelLimit>0)&&(AccLimit<=20 &&
AccLimit>0)){
                                        RotArray[1]->VelRotInt=StrToFloat(VelRot->Text);
                                        RotArray[1]->AccRotInt=StrToFloat(AccRot->Text);
                                        RotArray[1]->PosRotInt=StrToFloat(PosRot->Text);
                                        //Making sure that an order has been selected.
```

```
                                        if(RotOrdRG->ItemIndex!=-1){
                                                RotArray[1]->OrderRot=RotOrdRG-
>ItemIndex;
                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosRot->Modified || VelRot->Modified
|| AccRot->Modified){
                                                        RotParamPos[1] = true;

                                                }else{
                                                        if(RunButton){
                                                                RotParamPos[1] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                RotParamPos[1] = false;
                                                Application->MessageBox("Must select and
order for the rotational axis","Error",MB_OK);
                                                break;
                                        }

                                        PosRot->Modified = false;
                                }else{
                                        Application->MessageBox("Rotational velocity and
acceleration value must be between 0 - 20","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Vertical Axis
```

```
                        if(VelVert->Text!="" && AccVert->Text!="" && PosVert->Text!=""){
                                VelLimit = StrToFloat(VelVert->Text);
                                AccLimit = StrToFloat(AccVert->Text);
                                if((VelLimit<=15 && VelLimit>0)&&(AccLimit<=15 &&
AccLimit>0)){
                                        VertArray[1]->VelVertInt=StrToFloat(VelVert-
>Text);
                                        VertArray[1]->AccVertInt=StrToFloat(AccVert-
>Text);
                                        VertArray[1]->PosVertInt=StrToFloat(PosVert-
>Text);
                                        //Making sure that an order has been selected.
                                        if(VertOrdRG->ItemIndex!=-1){
                                                VertArray[1]->OrderVert=VertOrdRG-
>ItemIndex;
                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosVert->Modified || VelVert->Modified
|| AccVert->Modified){
                                                        VertParamPos[1] = true;
                                                }else{
                                                        if(RunButton){
                                                                VertParamPos[1] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                VertParamPos[1] = false;
```

```
                                                Application->MessageBox("Must select and
order for the vertical axis","Error",MB_OK);
                                                break;
                                        }

                                        PosVert->Modified = false;
                                }else{
                                        Application->MessageBox("Vertical velocity and
acceleration value must be between 0 - 15","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Gripper Axis
                        if(VelGrip->Text!="" && AccGrip->Text!="" && PosGrip->Text!=""){
                                VelLimit = StrToFloat(VelGrip->Text);
                                AccLimit = StrToFloat(AccGrip->Text);
                                //Check and see if vel and acc are with min and max
limits.
                                if((VelLimit<=40 && VelLimit>0)&&(AccLimit<=80 &&
AccLimit>0)){
                                        GripperArray[1]->VelGripInt=StrToFloat(VelGrip-
>Text);
                                        GripperArray[1]->AccGripInt=StrToFloat(AccGrip-
>Text);
                                        GripperArray[1]->PosGripInt=StrToFloat(PosGrip-
>Text);
                                        //Making sure that an order has been selected.
                                        if(GripOrdRG->ItemIndex!=-1){
                                                GripperArray[1]->OrderGrip=GripOrdRG-
>ItemIndex;
                                                ParamEntered = true;
```

```
                                                //Making sure that the position has been
altered.
                                                if(PosGrip->Modified || VelGrip->Modified
|| AccGrip->Modified){
                                                        GripParamPos[1] = true;

                                                }else{
                                                        if(RunButton){
                                                                GripParamPos[1] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                GripParamPos[1] = false;
                                                Application->MessageBox("Must select and
order for the gripper axis","Error",MB_OK);
                                                break;
                                        }

                                        PosGrip->Modified = false;
                                }else{
                                        Application->MessageBox("Gripper velocity value
must be between 0 - 40 and acceleration between 0 - 80.","Value Error",MB_OK);
                                        break;
                                }
                        }
                        gripoc[1] = GripOC->ItemIndex;
                        Enter = true;
                        break;

                        //Next button has been clicked twice
```

```
                        case 2:
                        //Horizontal Axis
                        if(VelHorz->Text!="" && AccHorz->Text!="" && PosHorz->Text!=""){
                                VelLimit = StrToFloat(VelHorz->Text);
                                AccLimit = StrToFloat(AccHorz->Text);
                                if((VelLimit<=15 && VelLimit>0)&&(AccLimit<=15 &&
AccLimit>0)){
                                        HorzArray[2]->VelHorzInt=StrToFloat(VelHorz-
>Text);
                                        HorzArray[2]->AccHorzInt=StrToFloat(AccHorz-
>Text);
                                        HorzArray[2]->PosHorzInt=StrToFloat(PosHorz-
>Text);
                                        //Making sure that an order has been selected.
                                        if(HorzOrdRG->ItemIndex!=-1){
                                                HorzArray[2]->OrderHorz=HorzOrdRG-
>ItemIndex;
                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosHorz->Modified || VelHorz->Modified
|| AccHorz->Modified){
                                                        HorzParamPos[2] = true;
                                                }else{
                                                        if(RunButton){
                                                                HorzParamPos[2] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
```

```
                                                HorzParamPos[2] = false;
                                                Application->MessageBox("Must select and
order for the horizontal axis","Error",MB_OK);
                                                break;
                                        }

                                        PosHorz->Modified = false;
                                }else{
                                        Application->MessageBox("Horizontal velocity and
acceleration value must be between 0 - 15","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Rotation Axis
                        if(VelRot->Text!="" && AccRot->Text!="" && PosRot->Text!=""){
                                VelLimit = StrToFloat(VelRot->Text);
                                AccLimit = StrToFloat(AccRot->Text);
                                if((VelLimit<=20 && VelLimit>0)&&(AccLimit<=20 &&
AccLimit>0)){
                                        RotArray[2]->VelRotInt=StrToFloat(VelRot->Text);
                                        RotArray[2]->AccRotInt=StrToFloat(AccRot->Text);
                                        RotArray[2]->PosRotInt=StrToFloat(PosRot->Text);
                                        //Making sure that an order has been selected.
                                        if(RotOrdRG->ItemIndex!=-1){
                                                RotArray[2]->OrderRot=RotOrdRG-
>ItemIndex;

                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosRot->Modified || VelRot->Modified
|| AccRot->Modified){
```

```
                                                RotParamPos[2] = true;

                                        }else{
                                                if(RunButton){
                                                        RotParamPos[2] = false;
                                                }
                                        }
                                }else{
                                        ParamEntered = false;
                                        RotParamPos[2] = false;
                                        Application->MessageBox("Must select and
order for the rotational axis","Error",MB_OK);
                                        break;
                                }

                                PosRot->Modified = false;
                        }else{
                                Application->MessageBox("Rotational velocity and
acceleration value must be between 0 - 20","Value Error",MB_OK);
                                break;
                        }
                }
                //Vertical Axis
                if(VelVert->Text!="" && AccVert->Text!="" && PosVert->Text!=""){
                        VelLimit = StrToFloat(VelVert->Text);
                        AccLimit = StrToFloat(AccVert->Text);
                        if((VelLimit<=15 && VelLimit>0)&&(AccLimit<=15 &&
AccLimit>0)){
                                VertArray[2]->VelVertInt=StrToFloat(VelVert-
>Text);
```

```
                                        VertArray[2]->AccVertInt=StrToFloat(AccVert-
>Text);
                                        VertArray[2]->PosVertInt=StrToFloat(PosVert-
>Text);
                                        //Making sure that an order has been selected.
                                        if(VertOrdRG->ItemIndex!=-1){
                                                VertArray[2]->OrderVert=VertOrdRG-
>ItemIndex;
                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosVert->Modified || VelVert->Modified
|| AccVert->Modified){
                                                        VertParamPos[2] = true;

                                                }else{
                                                        if(RunButton){
                                                                VertParamPos[2] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                VertParamPos[2] = false;
                                                Application->MessageBox("Must select and
order for the vertical axis","Error",MB_OK);
                                                break;
                                        }

                                        PosVert->Modified = false;
                                }else{
```

```
                                        Application->MessageBox("Vertical velocity and
acceleration value must be between 0 - 15","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Gripper Axis
                        if(VelGrip->Text!="" && AccGrip->Text!="" && PosGrip->Text!=""){
                                VelLimit = StrToFloat(VelGrip->Text);
                                AccLimit = StrToFloat(AccGrip->Text);
                                //Check and see if vel and acc are with min and max
limits.
                                if((VelLimit<=40 && VelLimit>0)&&(AccLimit<=80 &&
AccLimit>0)){
                                        GripperArray[2]->VelGripInt=StrToFloat(VelGrip-
>Text);
                                        GripperArray[2]->AccGripInt=StrToFloat(AccGrip-
>Text);
                                        GripperArray[2]->PosGripInt=StrToFloat(PosGrip-
>Text);
                                        //Making sure that an order has been selected.
                                        if(GripOrdRG->ItemIndex!=-1){
                                                GripperArray[2]->OrderGrip=GripOrdRG-
>ItemIndex;
                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosGrip->Modified || VelGrip->Modified
|| AccGrip->Modified){
                                                        GripParamPos[2] = true;
                                                }else{
```

```
                                                        if(RunButton){
                                                                GripParamPos[2] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                GripParamPos[2] = false;
                                                Application->MessageBox("Must select and
order for the gripper axis","Error",MB_OK);
                                                break;
                                        }

                                        PosGrip->Modified = false;

                                }else{
                                        Application->MessageBox("Gripper velocity value
must be between 0 - 40 and acceleration between 0 - 80.","Value Error",MB_OK);
                                        break;
                                }
                        }
                        gripoc[2] = GripOC->ItemIndex;
                        Enter = true;
                        break;

                       //Next button has been clicked three times (last time)
                        case 3:
                        //Horizontal Axis
                        if(VelHorz->Text!="" && AccHorz->Text!="" && PosHorz->Text!=""){
                                VelLimit = StrToFloat(VelHorz->Text);
                                AccLimit = StrToFloat(AccHorz->Text);
```

```
                                if((VelLimit<=15 && VelLimit>0)&&(AccLimit<=15 &&
AccLimit>0)){
                                        HorzArray[3]->VelHorzInt=StrToFloat(VelHorz-
>Text);
                                        HorzArray[3]->AccHorzInt=StrToFloat(AccHorz-
>Text);
                                        HorzArray[3]->PosHorzInt=StrToFloat(PosHorz-
>Text);
                                        //Making sure that an order has been selected.
                                        if(HorzOrdRG->ItemIndex!=-1){
                                                HorzArray[3]->OrderHorz=HorzOrdRG-
>ItemIndex;
                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosHorz->Modified || VelHorz->Modified
|| AccHorz->Modified){
                                                        HorzParamPos[3] = true;
                                                }else{
                                                        if(RunButton){
                                                                HorzParamPos[3] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                HorzParamPos[3] = false;
                                                Application->MessageBox("Must select and
order for the horizontal axis","Error",MB_OK);
                                                break;
                                        }
```

```
                                        PosHorz->Modified = false;
                                }else{
                                        Application->MessageBox("Horizontal velocity and
acceleration value must be between 0 - 15","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Rotation Axis
                        if(VelRot->Text!="" && AccRot->Text!="" && PosRot->Text!=""){
                                VelLimit = StrToFloat(VelRot->Text);
                                AccLimit = StrToFloat(AccRot->Text);
                                if((VelLimit<=20 && VelLimit>0)&&(AccLimit<=20 &&
AccLimit>0)){
                                        RotArray[3]->VelRotInt=StrToFloat(VelRot->Text);
                                        RotArray[3]->AccRotInt=StrToFloat(AccRot->Text);
                                        RotArray[3]->PosRotInt=StrToFloat(PosRot->Text);
                                        //Making sure that an order has been selected.
                                        if(RotOrdRG->ItemIndex!=-1){
                                                RotArray[3]->OrderRot=RotOrdRG-
>ItemIndex;

                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosRot->Modified || VelRot->Modified
|| AccRot->Modified){
                                                        RotParamPos[3] = true;

                                                }else{
                                                        if(RunButton){
                                                                RotParamPos[3] = false;
```

```
                                                }
                                        }
                                }else{
                                        ParamEntered = false;
                                        RotParamPos[3] = false;
                                        Application->MessageBox("Must select and
order for the rotational axis","Error",MB_OK);
                                        break;
                                }

                                PosRot->Modified = false;
                        }else{
                                Application->MessageBox("Rotational velocity and
acceleration value must be between 0 - 20","Value Error",MB_OK);
                                break;
                        }
                }
                //Vertical Axis
                if(VelVert->Text!="" && AccVert->Text!="" && PosVert->Text!=""){
                        VelLimit = StrToFloat(VelVert->Text);
                        AccLimit = StrToFloat(AccVert->Text);
                        if((VelLimit<=15 && VelLimit>0)&&(AccLimit<=15 &&
AccLimit>0)){
                                VertArray[3]->VelVertInt=StrToFloat(VelVert-
>Text);
                                VertArray[3]->AccVertInt=StrToFloat(AccVert-
>Text);
                                VertArray[3]->PosVertInt=StrToFloat(PosVert-
>Text);
                                //Making sure that an order has been selected.
                                if(VertOrdRG->ItemIndex!=-1){
```

```
                                                VertArray[3]->OrderVert=VertOrdRG-
>ItemIndex;
                                                ParamEntered = true;
                                                //Making sure that the position has been
altered.
                                                if(PosVert->Modified || VelVert->Modified
|| AccVert->Modified){
                                                        VertParamPos[3] = true;

                                                }else{
                                                        if(RunButton){
                                                                VertParamPos[3] = false;
                                                        }
                                                }
                                        }else{
                                                ParamEntered = false;
                                                VertParamPos[3] = false;
                                                Application->MessageBox("Must select and
order for the vertical axis","Error",MB_OK);
                                                break;
                                        }

                                        PosVert->Modified = false;
                                }else{
                                        Application->MessageBox("Vertical velocity and
acceleration value must be between 0 - 15","Value Error",MB_OK);
                                        break;
                                }
                        }
                        //Gripper Axis
                        if(VelGrip->Text!="" && AccGrip->Text!="" && PosGrip->Text!=""){
```

```
                        VelLimit = StrToFloat(VelGrip->Text);
                        AccLimit = StrToFloat(AccGrip->Text);
                        //Check and see if vel and acc are with min and max
limits.
                        if((VelLimit<=40 && VelLimit>0)&&(AccLimit<=80 &&
AccLimit>0)){
                                GripperArray[3]->VelGripInt=StrToFloat(VelGrip-
>Text);
                                GripperArray[3]->AccGripInt=StrToFloat(AccGrip-
>Text);
                                GripperArray[3]->PosGripInt=StrToFloat(PosGrip-
>Text);
                                //Making sure that an order has been selected.
                                if(GripOrdRG->ItemIndex!=-1){
                                        GripperArray[3]->OrderGrip=GripOrdRG-
>ItemIndex;
                                        ParamEntered = true;
                                        //Making sure that the position has been
altered.
                                        if(PosGrip->Modified || VelGrip->Modified
|| AccGrip->Modified){
                                                GripParamPos[3] = true;
                                        }else{
                                                if(RunButton){
                                                        GripParamPos[3] = false;
                                                }
                                        }
                                }else{
                                        ParamEntered = false;
                                        GripParamPos[3] = false;
```

```
                                                Application->MessageBox("Must select and
order for the gripper axis","Error",MB_OK);
                                                break;
                                        }

                                        PosGrip->Modified = false;
                                }else{
                                        Application->MessageBox("Gripper velocity value
must be between 0 - 40 and acceleration between 0 - 80.","Value Error",MB_OK);
                                        break;
                                }
                        }
                        Enter = true;
                        gripoc[3] = GripOC->ItemIndex;
                        break;
                }
        }
        //Making sure at least one axis had all values entered before proceeding.
        if(!ParamEntered){
                Application->MessageBox("At least one axis must have all values set
before pressing enter","Value Error",MB_OK);
                Enter = false;
        }
        OpenCheck = false;
        ParamEntered = false;

}
//---------------------------------------------------------------------------

void __fastcall Tmotion::NextbutClick(TObject *Sender)
{
```

```
if(next<3){
        next++;
        if(Enter){
                switch(next){
                        //Next has been pressed once
                        case 1:
                        if(HorzArray[1]->VelHorzInt && HorzArray[1]->AccHorzInt){
                                VelHorz->Text=HorzArray[1]->VelHorzInt;
                                AccHorz->Text=HorzArray[1]->AccHorzInt;
                                PosHorz->Text=HorzArray[1]->PosHorzInt;
                                HorzOrdRG->ItemIndex=HorzArray[1]->OrderHorz;
                        }else{
                                VelHorz->Clear();
                                AccHorz->Clear();
                                PosHorz->Clear();
                                HorzOrdRG->ItemIndex=-1;
                                HorzParamPos[1] = false;
                        }
                        if(RotArray[1]->VelRotInt && RotArray[1]->AccRotInt){
                                VelRot->Text=RotArray[1]->VelRotInt;
                                AccRot->Text=RotArray[1]->AccRotInt;
                                PosRot->Text=RotArray[1]->PosRotInt;
                                RotOrdRG->ItemIndex=RotArray[1]->OrderRot;
                        }else{
                                VelRot->Clear();
                                AccRot->Clear();
                                PosRot->Clear();
                                RotOrdRG->ItemIndex=-1;
                                RotParamPos[1] = false;
                        }
```

```
                        if((VertArray[1]->VelVertInt && VertArray[1]-
>AccVertInt)||CalMenuCheck){
                                VelVert->Text=VertArray[1]->VelVertInt;
                                AccVert->Text=VertArray[1]->AccVertInt;
                                PosVert->Text=VertArray[1]->PosVertInt;
                                VertOrdRG->ItemIndex=VertArray[1]->OrderVert;
                        }else{
                                VelVert->Clear();
                                AccVert->Clear();
                                PosVert->Clear();
                                VertOrdRG->ItemIndex=-1;
                                VertParamPos[1] = false;
                        }
                        if((GripperArray[1]->VelGripInt && GripperArray[1]-
>AccGripInt)||CalMenuCheck){
                                VelGrip->Text=GripperArray[1]->VelGripInt;
                                AccGrip->Text=GripperArray[1]->AccGripInt;
                                PosGrip->Text=GripperArray[1]->PosGripInt;
                                GripOrdRG->ItemIndex=GripperArray[1]->OrderGrip;
                        }else{
                                VelGrip->Clear();
                                AccGrip->Clear();
                                PosGrip->Clear();
                                GripOrdRG->ItemIndex=-1;
                                GripParamPos[1] = false;
                        }
                        GripOC->ItemIndex=gripoc[1];

                        //Coloring the motion number circles to the correct
color.
                        MoveNo1->Brush->Color=clWhite;
```

```
                                MoveNo2->Brush->Color=clGreen;
                                MoveNo3->Brush->Color=clWhite;
                                MoveNo4->Brush->Color=clWhite;

if(!(HorzParamPos[1]||RotParamPos[1]||VertParamPos[1]||GripParamPos[1])){
                                        Enter = false;
                                }
                                break;

                                case 2:
                                if(HorzArray[2]->VelHorzInt && HorzArray[2]->AccHorzInt){
                                        VelHorz->Text=HorzArray[2]->VelHorzInt;
                                        AccHorz->Text=HorzArray[2]->AccHorzInt;
                                        PosHorz->Text=HorzArray[2]->PosHorzInt;
                                        HorzOrdRG->ItemIndex=HorzArray[2]->OrderHorz;
                                }else{
                                        VelHorz->Clear();
                                        AccHorz->Clear();
                                        PosHorz->Clear();
                                        HorzOrdRG->ItemIndex=-1;
                                        HorzParamPos[2] = false;
                                }
                                if(RotArray[2]->VelRotInt && RotArray[2]->AccRotInt){
                                        VelRot->Text=RotArray[2]->VelRotInt;
                                        AccRot->Text=RotArray[2]->AccRotInt;
                                        PosRot->Text=RotArray[2]->PosRotInt;
                                        RotOrdRG->ItemIndex=RotArray[2]->OrderRot;
                                }else{
                                        VelRot->Clear();
                                        AccRot->Clear();
                                        PosRot->Clear();
```

```
                                        RotOrdRG->ItemIndex=-1;
                                        RotParamPos[2] = false;
                                }
                                if((VertArray[2]->VelVertInt && VertArray[2]-
>AccVertInt)||CalMenuCheck){
                                        VelVert->Text=VertArray[2]->VelVertInt;
                                        AccVert->Text=VertArray[2]->AccVertInt;
                                        PosVert->Text=VertArray[2]->PosVertInt;
                                        VertOrdRG->ItemIndex=VertArray[2]->OrderVert;
                                }else{
                                        VelVert->Clear();
                                        AccVert->Clear();
                                        PosVert->Clear();
                                        VertOrdRG->ItemIndex=-1;
                                        VertParamPos[2] = false;
                                }
                                if(GripperArray[2]->VelGripInt && GripperArray[2]-
>AccGripInt){
                                        VelGrip->Text=GripperArray[2]->VelGripInt;
                                        AccGrip->Text=GripperArray[2]->AccGripInt;
                                        PosGrip->Text=GripperArray[2]->PosGripInt;
                                        GripOrdRG->ItemIndex=GripperArray[2]->OrderGrip;
                                }else{
                                        VelGrip->Clear();
                                        AccGrip->Clear();
                                        PosGrip->Clear();
                                        GripOrdRG->ItemIndex=-1;
                                        GripParamPos[2] = false;
                                }
                                GripOC->ItemIndex=gripoc[2];
```

```
                                //Coloring the motion number circles to the correct
color.
                                MoveNo1->Brush->Color=clWhite;
                                MoveNo2->Brush->Color=clWhite;
                                MoveNo3->Brush->Color=clGreen;
                                MoveNo4->Brush->Color=clWhite;

if(!(HorzParamPos[2]||RotParamPos[2]||VertParamPos[2]||GripParamPos[2])){
                                        Enter = false;
                                }
                                break;
                                case 3:
                                if((HorzArray[3]->VelHorzInt && HorzArray[3]-
>AccHorzInt)||CalMenuCheck){
                                        VelHorz->Text=HorzArray[3]->VelHorzInt;
                                        AccHorz->Text=HorzArray[3]->AccHorzInt;
                                        PosHorz->Text=HorzArray[3]->PosHorzInt;
                                        HorzOrdRG->ItemIndex=HorzArray[3]->OrderHorz;
                                }else{
                                        VelHorz->Clear();
                                        AccHorz->Clear();
                                        PosHorz->Clear();
                                        HorzOrdRG->ItemIndex=-1;
                                        HorzParamPos[3] = false;
                                }
                                if(RotArray[3]->VelRotInt && RotArray[3]->AccRotInt){
                                        VelRot->Text=RotArray[3]->VelRotInt;
                                        AccRot->Text=RotArray[3]->AccRotInt;
                                        PosRot->Text=RotArray[3]->PosRotInt;
                                        RotOrdRG->ItemIndex=RotArray[3]->OrderRot;
                                }else{
```

```
                                        VelRot->Clear();
                                        AccRot->Clear();
                                        PosRot->Clear();
                                        RotOrdRG->ItemIndex=-1;
                                        RotParamPos[3] = false;
                                }
                                if((VertArray[3]->VelVertInt && VertArray[3]-
>AccVertInt)||CalMenuCheck){
                                        VelVert->Text=VertArray[3]->VelVertInt;
                                        AccVert->Text=VertArray[3]->AccVertInt;
                                        PosVert->Text=VertArray[3]->PosVertInt;
                                        VertOrdRG->ItemIndex=VertArray[3]->OrderVert;
                                }else{
                                        VelVert->Clear();
                                        AccVert->Clear();
                                        PosVert->Clear();
                                        VertOrdRG->ItemIndex=-1;
                                        VertParamPos[3] = false;
                                }
                                if((GripperArray[3]->VelGripInt && GripperArray[3]-
>AccGripInt)||CalMenuCheck){
                                        VelGrip->Text=GripperArray[3]->VelGripInt;
                                        AccGrip->Text=GripperArray[3]->AccGripInt;
                                        PosGrip->Text=GripperArray[3]->PosGripInt;
                                        GripOrdRG->ItemIndex=GripperArray[3]->OrderGrip;
                                }else{
                                        VelGrip->Clear();
                                        AccGrip->Clear();
                                        PosGrip->Clear();
                                        GripOrdRG->ItemIndex=-1;
                                        GripParamPos[3] = false;
```

```
                                }
                                GripOC->ItemIndex=gripoc[3];

                                //Coloring the motion number circles to the correct
color.
                                MoveNo1->Brush->Color=clWhite;
                                MoveNo2->Brush->Color=clWhite;
                                MoveNo3->Brush->Color=clWhite;
                                MoveNo4->Brush->Color=clGreen;

if(!(HorzParamPos[3]||RotParamPos[3]||VertParamPos[3]||GripParamPos[3])){
                                        Enter = false;
                                }
                                break;
                        }
                }
                else{
                        Application->MessageBox("Press enter to store parameters before
going to the next move.","Sequence Error",MB_OK);
                        next--;
                }
        }
        else{
                Application->MessageBox("Input limit for the number of moves has been
reached.","Message",MB_OK);
        }
}
//---------------------------------------------------------------------------
```

```
void __fastcall Tmotion::PrevbutClick(TObject *Sender)
{

        if(next>0){
                next--;
                switch(next){
                        case 0:
                        if(HorzArray[0]->VelHorzInt && HorzArray[0]->AccHorzInt){
                                VelHorz->Text=HorzArray[0]->VelHorzInt;
                                AccHorz->Text=HorzArray[0]->AccHorzInt;
                                PosHorz->Text=HorzArray[0]->PosHorzInt;
                                HorzOrdRG->ItemIndex=HorzArray[0]->OrderHorz;
                        }else{
                                VelHorz->Clear();
                                AccHorz->Clear();
                                PosHorz->Clear();
                                HorzOrdRG->ItemIndex=-1;
                        }
                        if(RotArray[0]->VelRotInt && RotArray[0]->AccRotInt){
                                VelRot->Text=RotArray[0]->VelRotInt;
                                AccRot->Text=RotArray[0]->AccRotInt;
                                PosRot->Text=RotArray[0]->PosRotInt;
                                RotOrdRG->ItemIndex=RotArray[0]->OrderRot;
                        }else{
                                VelRot->Clear();
                                AccRot->Clear();
                                PosRot->Clear();
                                RotOrdRG->ItemIndex=-1;
                        }
                        if(VertArray[0]->VelVertInt && VertArray[0]->AccVertInt){
                                VelVert->Text=VertArray[0]->VelVertInt;
```

```
        AccVert->Text=VertArray[0]->AccVertInt;
        PosVert->Text=VertArray[0]->PosVertInt;
        VertOrdRG->ItemIndex=VertArray[0]->OrderVert;
}else{
        VelVert->Clear();
        AccVert->Clear();
        PosVert->Clear();
        VertOrdRG->ItemIndex=-1;
}
if(GripperArray[0]->VelGripInt && GripperArray[0]->AccGripInt){
        VelGrip->Text=GripperArray[0]->VelGripInt;
        AccGrip->Text=GripperArray[0]->AccGripInt;
        PosGrip->Text=GripperArray[0]->PosGripInt;
        GripOrdRG->ItemIndex=GripperArray[0]->OrderGrip;
}else{
        VelGrip->Clear();
        AccGrip->Clear();
        PosGrip->Clear();
        GripOrdRG->ItemIndex=-1;
}
GripOC->ItemIndex=gripoc[0];

//Coloring the motion number circles to the correct color.
MoveNo1->Brush->Color=clGreen;
MoveNo2->Brush->Color=clWhite;
MoveNo3->Brush->Color=clWhite;
MoveNo4->Brush->Color=clWhite;

break;

case 1:
```

```
if(HorzArray[1]->VelHorzInt && HorzArray[1]->AccHorzInt){
        VelHorz->Text=HorzArray[1]->VelHorzInt;
        AccHorz->Text=HorzArray[1]->AccHorzInt;
        PosHorz->Text=HorzArray[1]->PosHorzInt;
        HorzOrdRG->ItemIndex=HorzArray[1]->OrderHorz;
}else{
        VelHorz->Clear();
        AccHorz->Clear();
        PosHorz->Clear();
        HorzOrdRG->ItemIndex=-1;
}
if(RotArray[1]->VelRotInt && RotArray[1]->AccRotInt){
        VelRot->Text=RotArray[1]->VelRotInt;
        AccRot->Text=RotArray[1]->AccRotInt;
        PosRot->Text=RotArray[1]->PosRotInt;
        RotOrdRG->ItemIndex=RotArray[1]->OrderRot;
}else{
        VelRot->Clear();
        AccRot->Clear();
        PosRot->Clear();
        RotOrdRG->ItemIndex=-1;
}
if(VertArray[1]->VelVertInt && VertArray[1]->AccVertInt){
        VelVert->Text=VertArray[1]->VelVertInt;
        AccVert->Text=VertArray[1]->AccVertInt;
        PosVert->Text=VertArray[1]->PosVertInt;
        VertOrdRG->ItemIndex=VertArray[1]->OrderVert;
}else{
        VelVert->Clear();
        AccVert->Clear();
        PosVert->Clear();
```

```
        VertOrdRG->ItemIndex=-1;
}
if(GripperArray[1]->VelGripInt && GripperArray[1]->AccGripInt){
        VelGrip->Text=GripperArray[1]->VelGripInt;
        AccGrip->Text=GripperArray[1]->AccGripInt;
        PosGrip->Text=GripperArray[1]->PosGripInt;
        GripOrdRG->ItemIndex=GripperArray[1]->OrderGrip;
}else{
        VelGrip->Clear();
        AccGrip->Clear();
        PosGrip->Clear();
        GripOrdRG->ItemIndex=-1;
}
GripOC->ItemIndex=gripoc[1];

//Coloring the motion number circles to the correct color.
MoveNo1->Brush->Color=clWhite;
MoveNo2->Brush->Color=clGreen;
MoveNo3->Brush->Color=clWhite;
MoveNo4->Brush->Color=clWhite;

break;

case 2:
if(HorzArray[2]->VelHorzInt && HorzArray[2]->AccHorzInt){
        VelHorz->Text=HorzArray[2]->VelHorzInt;
        AccHorz->Text=HorzArray[2]->AccHorzInt;
        PosHorz->Text=HorzArray[2]->PosHorzInt;
        HorzOrdRG->ItemIndex=HorzArray[2]->OrderHorz;
}else{
        VelHorz->Clear();
```

```
        AccHorz->Clear();
        PosHorz->Clear();
        HorzOrdRG->ItemIndex=-1;
}
if(RotArray[2]->VelRotInt && RotArray[2]->AccRotInt){
        VelRot->Text=RotArray[2]->VelRotInt;
        AccRot->Text=RotArray[2]->AccRotInt;
        PosRot->Text=RotArray[2]->PosRotInt;
        RotOrdRG->ItemIndex=RotArray[2]->OrderRot;
}else{
        VelRot->Clear();
        AccRot->Clear();
        PosRot->Clear();
        RotOrdRG->ItemIndex=-1;
}
if(VertArray[2]->VelVertInt && VertArray[2]->AccVertInt){
        VelVert->Text=VertArray[2]->VelVertInt;
        AccVert->Text=VertArray[2]->AccVertInt;
        PosVert->Text=VertArray[2]->PosVertInt;
        VertOrdRG->ItemIndex=VertArray[2]->OrderVert;
}else{
        VelVert->Clear();
        AccVert->Clear();
        PosVert->Clear();
        VertOrdRG->ItemIndex=-1;
}
if(GripperArray[2]->VelGripInt && GripperArray[2]->AccGripInt){
        VelGrip->Text=GripperArray[2]->VelGripInt;
        AccGrip->Text=GripperArray[2]->AccGripInt;
        PosGrip->Text=GripperArray[2]->PosGripInt;
        GripOrdRG->ItemIndex=GripperArray[2]->OrderGrip;
```

```
                        }else{
                                VelGrip->Clear();
                                AccGrip->Clear();
                                PosGrip->Clear();
                                GripOrdRG->ItemIndex=-1;
                        }
                        GripOC->ItemIndex=gripoc[2];

                        //Coloring the motion number circles to the correct color.
                        MoveNo1->Brush->Color=clWhite;
                        MoveNo2->Brush->Color=clWhite;
                        MoveNo3->Brush->Color=clGreen;
                        MoveNo4->Brush->Color=clWhite;

                        break;

                }
                Enter=true;
        }
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::ResbutClick(TObject *Sender)
{
        for(int i=0;i<4;i++){
                delete HorzArray[i];
                delete VertArray[i];
                delete RotArray[i];
                delete GripperArray[i];
                gripoc[i] = -1;
```

```
}

for(int i=0;i<4;i++){
HorzArray[i] = new HorizontalAxis();
VertArray[i] = new VerticalAxis();
RotArray[i] = new RotationAxis();
GripperArray[i] = new GripperAxis();
}

RepeatHorz = false;
RepeatVert = false;
RepeatRot = false;
RepeatGrip = false;

//Clear the text boxes.
VelHorz->Clear();
AccHorz->Clear();
PosHorz->Clear();
VelVert->Clear();
AccVert->Clear();
PosVert->Clear();
VelRot->Clear();
AccRot->Clear();
PosRot->Clear();
PosGrip->Clear();
AccGrip->Clear();
VelGrip->Clear();
GripOC->ItemIndex=-1;
HorzOrdRG->ItemIndex=-1;
RotOrdRG->ItemIndex=-1;
VertOrdRG->ItemIndex=-1;
```

```
        GripOrdRG->ItemIndex=-1;
        next=0;
        Enter = false;
        ParamEntered = false;
        //Coloring the motion number circles to the correct color.
        MoveNo1->Brush->Color=clGreen;
        MoveNo2->Brush->Color=clWhite;
        MoveNo3->Brush->Color=clWhite;
        MoveNo4->Brush->Color=clWhite;


        HorzEnable->State = cbUnchecked;
        VertEnable->State = cbUnchecked;
        RotEnable->State = cbUnchecked;
        GripEnable->State = cbUnchecked;

        enableAxes = 0x1E;

        HorzEnableClick(Sender);
        VertEnableClick(Sender);
        RotEnableClick(Sender);
        GripEnableClick(Sender);

        ResetCheck = true;
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::RunButClick(TObject *Sender)
{
        axis = 1;
```

```
        status == flex_read_csr_rtn(boardID, &csr);
/*Checking to see if parameters have been entered into the edit boxes.  Converting them
to NI's format.  Loading the parameters to each axis.  Then sequentially execute each
move command.*/
        if(InitCheck && Enter){

                RepeatHorz = HorzRep->Checked;
                RepeatVert = VertRep->Checked;
                RepeatRot = RotRep->Checked;
                RepeatGrip = GripRep->Checked;
                HaltCheck = false;
                AxisStatus->Visible = true;
                MoveComplete->Resume();
        }
        else{
                Application->MessageBox("Please initialize the board before
moving.","Error",MB_OK);
        }

        RunButton = true;
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::StopButClick(TObject *Sender)
{
        if(status == NIMC_noError){
                status = flex_stop_motion(boardID, 0, NIMC_DECEL_STOP, Axes1_4);
        }
}
//---------------------------------------------------------------------------
//My defined function to initialize the board. Prototype in Tinmotion class.
```

```
void Tmotion::InitializeBoard(TObject *Sender)
{
        bool HomeCheck=false;           //Home switch found check for axes 1, 3, and 4.
        bool IndexCheck=false;          //Index found check for all axes.
        accelDecel=20;
        velocity=200;
        targetPos=0;
        enableAxes=0;
        BPon=0x10;
        BPoff=0x0E;

        AxisStatus->Visible = false;
        //Implementing progress bar on form Initialize Board.
        InitBoard->PBar1->StepIt();
        InitBoard->InitStatusB->SimpleText="Initializing....";

        status = flex_clear_pu_status(boardID); //Clearing power up reset bit

        if(status == NIMC_noError){
                do{
                        status = flex_read_csr_rtn(boardID, &csr);
                        if(status != NIMC_noError) break;
                }while(csr & NIMC_POWER_UP_RESET);
        }

        //Setting breakpoint outputs off for axis 1-3 and on for axis 4
        if(status == NIMC_noError){
                status = flex_set_breakpoint_momo(boardID, 0, BPon, BPoff, HOST);
        }

        if(status==NIMC_noError){
```

```
                status = flex_enable_axes(boardID,NIMC_AXIS_CTRL,3,enableAxes);
        }

        //Map axis resources (encoder and DAC) to proper axes.
        for(axis=1;axis<5;axis++){

                if(status == NIMC_noError){
                        status = flex_config_axis(boardID,axis,primaryFBK,0,dac,0);
                }
                primaryFBK++;
                dac++;
        }

        primaryFBK=0x21;
        dac=0x31;

        InitBoard->PBar1->StepIt();

        for(axis=1;axis<5;axis++){
                //Loading Move complete criteria.
                if(status == NIMC_noError){
                        status =
flex_config_mc_criteria(boardID,axis,moveCriteria,700,10,10);
                }//Loading PID values for axis 1-3
                if(status == NIMC_noError){
                        status = flex_load_pid_parameters(boardID,axis,&PIDvalues,HOST);
                }//Loading PID values for axis 4
                if((status == NIMC_noError) && (axis == 4)){
                        status =
flex_load_single_pid_parameter(boardID,4,NIMC_KP,315,HOST);
```

```
                        status =
flex_load_single_pid_parameter(boardID,4,NIMC_KI,72,HOST);
                        status =
flex_load_single_pid_parameter(boardID,4,NIMC_KD,1904,HOST);
                        status =
flex_load_single_pid_parameter(boardID,4,NIMC_TD,3,HOST);
                }//Loading Counts per Revolution
                if(status == NIMC_noError){
                        status =
flex_load_counts_steps_rev(boardID,axis,NIMC_COUNTS,4000);
                }//Setting Operation Mode.
                if(status == NIMC_noError){
                        status = flex_set_op_mode(boardID,axis,NIMC_ABSOLUTE_POSITION);
                }//Setting Following Error to each axis.
                if(status == NIMC_noError){
                        status = flex_load_follow_err(boardID,axis,followingError,HOST);
                }//Load initial RPM value
                if(status == NIMC_noError){
                        status = flex_load_rpm(boardID,axis,velocity,HOST);
                }//Load initial acceleration value
                if(status == NIMC_noError){
                        status = flex_load_rpsps(boardID,axis,NIMC_BOTH,accelDecel,HOST);
                }
        }

        InitBoard->PBar1->StepIt();

        //Enabling Home input switches bitmap
        if(status == NIMC_noError){
                status=flex_enable_home_inputs(boardID,home);
        }//Enabling Hardware Limit switches for axes 1 and 3.
```

```
        if(status == NIMC_noError){

status=flex_enable_limits(boardID,NIMC_LIMIT_INPUTS,limitEnable,limitEnable);
        }//Enabling Software limit switch for axes 4.
        if(status == NIMC_noError){
                status=flex_enable_limits(boardID,NIMC_SOFTWARE_LIMITS,0x10,0x10);
        }//Setting software limit in counts for forward and reverse on axis 4
        if(status == NIMC_noError){
                status=flex_load_sw_lim_pos(boardID,4,100000,-100000,HOST);
        }//Setting Limit polarities
        if(status == NIMC_noError){
                status = flex_set_limit_polarity(boardID, limitPolarity,limitPolarity);
        }//Setting Home polarities
        if(status == NIMC_noError){
                status = flex_set_home_polarity(boardID,homePolarity);
        }//Enabling inhibit output for axis 4.
        if(status == NIMC_noError){
                status = flex_configure_inhibits(boardID, 0x10, 0x10);
        }

        InitBoard->PBar1->StepIt();

        findHome4 = 0x0001;

        //Enabling the Axes.

        enableAxes = 0x1E; //Setting axes 1-4 on.

        if(status == NIMC_noError){
                status = flex_enable_axes(boardID,NIMC_AXIS_CTRL,3,enableAxes);
        }
```

```
//Finding the home switch on axis 1.
if(status == NIMC_noError){
        status = flex_find_home(boardID,1,findHome1);
}//Finding the home switch on axis 3.
if(status == NIMC_noError){
        status = flex_find_home(boardID,3,findHome3);
}

InitBoard->InitStatusB->SimpleText="Finding Home....";

InitBoard->PBar1->StepIt();

do{
        //Checking the axis status
        if(status == NIMC_noError){
                status = flex_read_axis_status_rtn(boardID, 1, &AxisRtn1);
        }
        if(status == NIMC_noError){
                status = flex_read_axis_status_rtn(boardID, 3, &AxisRtn3);
        }
        //If there was a modal error report it.
        CheckModalError();

        //If there was a non-modal error break out of the loop.
        if(status != NIMC_noError) break;

        //Masking out the home found status bit of each axis and comparing them.
        HomeCheck = ((0x0400 & AxisRtn1) && (0x0400 & AxisRtn3));

//HomeCheck will equal zero until the home switch is found on all axes.
```

```
        }while(!HomeCheck);

        InitBoard->PBar1->StepIt();

        //Finding the index pulse and reseting Position.
        InitBoard->InitStatusB->SimpleText="Finding Index....";

        //Axis number 1.
       /* if(status == NIMC_noError){
                status = flex_find_index(boardID,1,0,0);
        }
        do{
                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 1, &AxisRtn1);
                }

                CheckModalError();

                if(errorCode != 0)break;

                IndexCheck = 0x0800 & AxisRtn1;

                if(status != NIMC_noError) break;

        }while(!IndexCheck);

        if(errorCode != 0){
                Application->MessageBox("Find Index Error. Please Reinitialize","Index
Error",MB_OK);
                errorCode=0;
        }
```

```
        //Axis number 2.
        if(status == NIMC_noError){
                status = flex_find_index(boardID,2,1,2500);
        }
        do{
                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 2, &AxisRtn2);
                }

                CheckModalError();

                if(errorCode != 0) break;

                IndexCheck = 0x0800 & AxisRtn2;

                if(status != NIMC_noError) break;

        }while(!IndexCheck);

        if(errorCode != 0){
                Application->MessageBox("Find Index Error. Please Reinitialize","Index
Error",MB_OK);
                errorCode=0;
        }

        //Axis number 3.
        if(status == NIMC_noError){
                status = flex_find_index(boardID,3,1,0);
        }
        do{
```

```
                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 3, &AxisRtn3);
                }

                CheckModalError();

                if(errorCode != 0) break;

                IndexCheck = 0x0800 & AxisRtn3;

                if(status != NIMC_noError) break;

        }while(!IndexCheck);

        if(errorCode != 0){
                Application->MessageBox("Find Index Error. Please Reinitialize","Index
Error",MB_OK);
                errorCode=0;
        }
        //Axis number 4.
        /*if(status == NIMC_noError){
                status = flex_find_index(boardID,4,1,0);
        }
        do{
                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 4, &AxisRtn4);
                }

                CheckModalError();

                if(errorCode != 0) break;
```

```
                IndexCheck = 0x0800 & AxisRtn4;

                if(status != NIMC_noError) break;

        }while(!IndexCheck);

        if(errorCode != 0){
                Application->MessageBox("Find Index Error. Please Reinitialize","Index
Error",MB_OK);
                errorCode=0;
        } */

        InitBoard->PBar1->StepIt();

        //Reseting position to zero.
        for(axis=1;axis<5;axis++){
                if(status == NIMC_noError){
                status = flex_reset_pos(boardID,axis,0,0,HOST);
                }
        }

        //Halt all axes movement
        if(status == NIMC_noError){
                status =
flex_stop_motion(boardID,NIMC_AXIS_CTRL,NIMC_HALT_STOP,enableAxes);
        }

        InitBoard->PBar1->StepIt();

        CheckModalError();
```

```
        InitBoard->PBar1->StepIt();

        //If there was a non-modal error report it.
        if (status != NIMC_noError){
          ErrorHandler(status, 0, 0);
        }
        axis=1;

        //Disabling Home input switches.
        if(status == NIMC_noError){
                status=flex_enable_home_inputs(boardID,0x0000);
        }

        InitBoard->PBar1->StepIt();

        if(status == NIMC_noError){
                InitCheck = true;
        }

        InitBoard->OkBut->Enabled = true;

        InitBoard->InitStatusB->SimpleText="Done.";

        ResbutClick(Sender);

}
//---------------------------------------------------------------------------
//Function to check for modal error message. Prototype in Tinmotion class.

void Tmotion::CheckModalError()
```

```
{

        if(status == NIMC_noError){
                status = flex_read_csr_rtn(boardID, &csr);
                if(status == NIMC_noError){
                        while(csr & NIMC_MODAL_ERROR_MSG){
                                status =
flex_read_error_msg_rtn(boardID,&commandID,&resource,&errorCode);
                                if(status != NIMC_noError) break;
                                ErrorHandler(errorCode,commandID,resource);
                                status = flex_read_csr_rtn(boardID,&csr);
                                if(status != NIMC_noError) break;
                        }
                }
        }
}
//-------------------------------------------------------------------------
//My function to load axes parameters.

void LoadParameters(u8 mAxis, f64 mVel, f64 mAccDecel, i32 mPos){

        status = flex_read_csr_rtn(boardID, &csr);
        //Loading parameters for each axis.
        if(status == NIMC_noError){
                status = flex_load_rpsps(boardID, mAxis, NIMC_BOTH, mAccDecel, HOST);
        }
        if(status == NIMC_noError){
                status = flex_load_rpm(boardID, mAxis, mVel, HOST);
        }
        if(status == NIMC_noError){
                status = flex_load_target_pos(boardID, mAxis, mPos, HOST);
```

```
        }
}
//---------------------------------------------------------------------------
void Tmotion::MoveAxes()
{
        status = flex_read_csr_rtn(boardID, &csr);

        if(status == NIMC_noError){
                status = flex_start(boardID, 0, Axes1_4);
        }
        //Report Non-modal error.
        if(status){
                ErrorHandler(status,0,0);
        }

        do{
                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 1, &AxisRtn1);
                }

                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 2, &AxisRtn2);
                }

                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 3, &AxisRtn3);
                }

                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 4, &AxisRtn4);
                }
```

```
                CheckModalError();

                //If any function to the FlexMotion board failed due to a
                //non-modal error exit the loop.
                if(status){
                        ErrorHandler(status, 0, 0);
                        break;
                }
                //Test against the move complete bit or the axis off bit
        }while(!((AxisRtn1 & (NIMC_MOVE_COMPLETE_BIT | NIMC_AXIS_OFF_BIT)) &&
                (AxisRtn2 & (NIMC_MOVE_COMPLETE_BIT | NIMC_AXIS_OFF_BIT)) &&
                (AxisRtn3 & (NIMC_MOVE_COMPLETE_BIT | NIMC_AXIS_OFF_BIT)) &&
                (AxisRtn4 & (NIMC_MOVE_COMPLETE_BIT | NIMC_AXIS_OFF_BIT))));
}
//---------------------------------------------------------------------------
//Function written by NI to record and print an Error.
void ErrorHandler(i32 errorCode, u16 commandID, u16 resource){

     i8 *errorDescription;         //Pointer to i8's -  to get error description
     u32 sizeOfArray;         //Size of error description
     u16 descriptionType;          //The type of description to be printed
     i32 status;              //Error returned by function

     if(commandID == 0){
          descriptionType = NIMC_ERROR_ONLY;
     }else{
          descriptionType = NIMC_COMBINED_DESCRIPTION;
     }

     //First get the size for the error description
```

```
     sizeOfArray = 0;
     errorDescription = NULL;//Setting this to NULL returns the size required
     status = flex_get_error_description(descriptionType, errorCode, commandID, resource,
                                                 errorDescription, &sizeOfArray );

     //Allocate memory on the heap for the description
     errorDescription = new i8[sizeOfArray + 1];

     sizeOfArray++; //So that the sizeOfArray is size of description + NULL character
     // Get Error Description
     status = flex_get_error_description(descriptionType, errorCode, commandID, resource,
                                                 errorDescription, &sizeOfArray );

     if (errorDescription != NULL){
          Application->MessageBox(errorDescription,"Error",MB_OK);
          free(errorDescription);//Free allocated memory
     }else{
          printf("Memory Allocation Error");
     }
        axis=1;
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::HorzEnableClick(TObject *Sender)
{

        if(HorzEnable->Checked){
                VelHorz->Enabled = true;
                VelHorz->Color = clWindow;
                AccHorz->Enabled = true;
                AccHorz->Color = clWindow;
```

```
                PosHorz->Enabled = true;
                PosHorz->Color = clWindow;
                enableAxes = enableAxes + 0x02;
        }
        else{
                VelHorz->Enabled = false;
                VelHorz->Color = clBtnFace;
                AccHorz->Enabled = false;
                AccHorz->Color = clBtnFace;
                PosHorz->Enabled = false;
                PosHorz->Color = clBtnFace;
                enableAxes = enableAxes - 0x02;
        }

}
//---------------------------------------------------------------------------

void __fastcall Tmotion::VertEnableClick(TObject *Sender)
{
        if(VertEnable->Checked){
                VelVert->Enabled = true;
                VelVert->Color = clWindow;
                AccVert->Enabled = true;
                AccVert->Color = clWindow;
                PosVert->Enabled = true;
                PosVert->Color = clWindow;
                enableAxes = enableAxes + 0x04;
        }
        else{
                VelVert->Enabled = false;
                VelVert->Color = clBtnFace;
```

```
                AccVert->Enabled = false;
                AccVert->Color = clBtnFace;
                PosVert->Enabled = false;
                PosVert->Color = clBtnFace;
                enableAxes = enableAxes - 0x04;
        }
}
//---------------------------------------------------------------------------

void __fastcall Tmotion::RotEnableClick(TObject *Sender)
{
        if(RotEnable->Checked){
                VelRot->Enabled = true;
                VelRot->Color = clWindow;
                AccRot->Enabled = true;
                AccRot->Color = clWindow;
                PosRot->Enabled = true;
                PosRot->Color = clWindow;
                enableAxes = enableAxes + 0x08;
        }
        else{
                VelRot->Enabled = false;
                VelRot->Color = clBtnFace;
                AccRot->Enabled = false;
                AccRot->Color = clBtnFace;
                PosRot->Enabled = false;
                PosRot->Color = clBtnFace;
                enableAxes = enableAxes - 0x08;
        }
}
//---------------------------------------------------------------------------
```

```
void __fastcall Tmotion::GripEnableClick(TObject *Sender)
{
        if(GripEnable->Checked){
                VelGrip->Enabled = true;
                VelGrip->Color = clWindow;
                AccGrip->Enabled = true;
                AccGrip->Color = clWindow;
                PosGrip->Enabled = true;
                PosGrip->Color = clWindow;

                GripOC->Enabled = true;
                GripCloseBtn->Enabled = true;
                GripOpenBtn->Enabled = true;
                enableAxes = enableAxes + 0x10;
        }
        else{
                VelGrip->Enabled = false;
                VelGrip->Color = clBtnFace;
                AccGrip->Enabled = false;
                AccGrip->Color = clBtnFace;
                PosGrip->Enabled = false;
                PosGrip->Color = clBtnFace;

                GripOC->Enabled = false;
                GripCloseBtn->Enabled = false;
                GripOpenBtn->Enabled = false;
                enableAxes = enableAxes - 0x10;
        }
}
//---------------------------------------------------------------------------
```

```
void __fastcall Tmotion::HomeButClick(TObject *Sender)
{
        //Move axes back to the zero postion.
        if(InitCheck){
                for(axis=1; axis<5; axis++){
                        LoadParameters(axis, 250, 20, 0);
                }
                MoveAxes();
        }
}
//---------------------------------------------------------------------------
void __fastcall Tmotion::DetectedButClick(TObject *Sender)
{
        int CenterPixelx = 37;
        int CenterPixely = 37;
        int HorzDisDes = 0, VertDisDes = 0, HorzDis = 0, VertDis = 0;
        int DesAngle = 0, ActAngle = 0;

        CamFuncCheck = MotionVideoOCX->CaptureToClipboard();

        CamFuncCheck = true;

        if(EnCapMotion){

                if(CamFuncCheck){
                        MotionPicture->Assign(Clipboard());

                        GetRawData(MotionPicture);
```

```
                        ColorFound = ColorFind(ColorLowerBound, ColorUpperBound,
ColorSelected);

                        if(ColorFound){

                                HorzDisDes = PixelxDes - CenterPixelx;
                                VertDisDes = CenterPixely - PixelyDes;

                                HorzDis = Pixelx - CenterPixelx;
                                VertDis = CenterPixely - Pixely;

                                Angle1 = atan2((double)VertDisDes, (double)HorzDisDes);

                                Angle2 = atan2((double)VertDis, (double)HorzDis);
                        }
                }
                else{
                        Application->MessageBoxA("Video Capture device failed to capture
image.","Error",MB_OK);
                }
                //Checking to see if angle 1 is in quad 3 or 4
                if(Angle1<0){
                        DesAngle = 360 + (180*Angle1)/M_PI;
                }
                else{
                        DesAngle = (180*Angle1)/M_PI;
                }
                //Checking to see if angle 2 is in quad 3 or 4
                if(Angle2<0){
                        ActAngle = 360 + (180*Angle2)/M_PI;
                }
```

```
                else{
                        ActAngle = (180*Angle2)/M_PI;
                }
                //Checking to see if angle 2 is less than angle 1 and in quad 1
                if(ActAngle<DesAngle && DesAngle<=90){
                        AngleDif = DesAngle - ActAngle;
                }
                else{
                        AngleDif = ActAngle - DesAngle;
                }
                //Checking to see if the difference of the angles is less than -180
                if(AngleDif<-180){
                        AngleDif = 360 + AngleDif;
                }
                //Checking to see if the difference of the angles is greater than 180
                if(AngleDif>180){
                        AngleDif = AngleDif - 360;
                }

                GripperArray[1]->PosGripInt = AngleDif;

        }
        else{
                Application->MessageBoxA("Enable vision feedback.","Error",MB_OK);
        }

}
//---------------------------------------------------------------------------
```

Initialize Board Form Code (Init_board.cpp)

```
//---------------------------------------------------------------------------

#include <vcl.h>
#include <io.h>
#pragma hdrstop

#include "FlexMotn.h"
#include "Init_board.h"
#include "inmotion.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"

extern i32 status;
TInitBoard *InitBoard;

//---------------------------------------------------------------------------
__fastcall TInitBoard::TInitBoard(TComponent* Owner)
        : TForm(Owner)
{

}
//---------------------------------------------------------------------------
void __fastcall TInitBoard::OkButClick(TObject *Sender)
{
Close();
}
//---------------------------------------------------------------------------

void __fastcall TInitBoard::NoButClick(TObject *Sender)
```

```
{
InitBoard->Visible=false;
}
//---------------------------------------------------------------------------


void __fastcall TInitBoard::YesButClick(TObject *Sender)
{
        PBar1->Step=10;
        motion->InitializeBoard(Sender);
        InitStatusB->SimpleText="Done";
}
//---------------------------------------------------------------------------
```

Axis Status Form Code (Axis_status.cpp)

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Axis_status.h"
#include "inmotion.h"
#include "Move.h"
#define HOST 0xFF

//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TAxisStatus *AxisStatus;

extern u16 Axes1_4;
extern u16 findHome1;
extern u16 findHome3;
extern u8 boardID;
extern u8 axis;
extern i32 status;
extern u16 AxisRtn1, AxisRtn2, AxisRtn3, AxisRtn4;
bool HaltCheck = false;

//---------------------------------------------------------------------------
__fastcall TAxisStatus::TAxisStatus(TComponent* Owner) : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TAxisStatus::CloseClick(TObject *Sender)
```

```
{
        AxisStatus->Visible=false;

}
//---------------------------------------------------------------------------
void __fastcall TAxisStatus::HaltMoveClick(TObject *Sender)
{
        if(status == NIMC_noError){
                status = flex_stop_motion(boardID,0,NIMC_DECEL_STOP,Axes1_4);
        }
        HaltCheck = true;
}
//---------------------------------------------------------------------------

void __fastcall TAxisStatus::ContinueButClick(TObject *Sender)
{
        motion->RunButClick(Sender);

}
//---------------------------------------------------------------------------
```

Threaded Application Code (Move.cpp)

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Move.h"
#include "inmotion.h"
#include "Axis_status.h"
#include "axesparam.h"
#define HOST 0xFF
#pragma package(smart_init)
//---------------------------------------------------------------------------
//Function Prototypes defined in imotion.cpp
//Errorhandler will print the error discription to screen
void ErrorHandler(i32 errorCode, u16 commandID, u16 resourceID);
//Function to load the parameters to each axis
void LoadParameters(u8 mAxis, f64 mAcc, f64 mVel, i32 mPos);
//---------------------------------------------------------------------------
int BrkPon=0x00;                //Breakpoint output variable to turn on in motion LED's
int BrkPoff=0x00;               //Breakpoint output vaiable to turn off in motion LED's
u16 MoveAxesOrder=0;            //Variable to hold axis move order
//External variables defined under another source.
extern i32 status;
extern u8 boardID;
extern u8 axis;
extern u16 AxisRtn1, AxisRtn2, AxisRtn3, AxisRtn4;
extern i32 positionRet[4];
extern i32 velocityRet[4];
extern bool HorzParamPos[4];
extern bool VertParamPos[4];
extern bool RotParamPos[4];
extern bool GripParamPos[4];
```

```
extern i32 targetPos;
extern f64 accelDecel;
extern f64 velocity;
extern u16 Axes1_4;
extern u16 csr;
extern int gripoc[4];
extern bool RepeatHorz;
extern bool RepeatVert;
extern bool RepeatRot;
extern bool RepeatGrip;
extern bool HaltCheck;
extern bool RunButton;
extern HorizontalAxis *HorzArray[4];
extern VerticalAxis *VertArray[4];
extern RotationAxis *RotArray[4];
extern GripperAxis *GripperArray[4];
//---------------------------------------------------------------------------
__fastcall MoveThread::MoveThread(bool CreateSuspended)
        : TThread(CreateSuspended)
{
        FreeOnTerminate = True;
        Priority = tpNormal;
}
//---------------------------------------------------------------------------
void __fastcall MoveThread::Execute()
{
        Axes1_4 = 0;
        axis = 1;
        u16 TriggerInputs;
        bool GripClose = false;
        bool GripOpen = false;
```

```
        RunButton = false;

        //Wait for move to complete and check for modal errors.
        while(!Terminated){
                AxisStatus->Close->Enabled = false;
                //Axis 1 move 1. Note lead on ball screw is .2"
                if(HorzParamPos[0]){
                        velocity = 118.11*(HorzArray[0]->VelHorzInt);
                        accelDecel = 1.97*(HorzArray[0]->AccHorzInt);
                        targetPos = 7874.02*(HorzArray[0]->PosHorzInt);
                        LoadParameters(axis, velocity, accelDecel, targetPos);
                        Axes1_4 = Axes1_4 + 0x0002; //Setting bitmap of axes to include
axis 1
                }
                axis++;
                //Axis 2 move 1. Note worm gear ratio of 180:1
                if(RotParamPos[0]){
                        velocity = 30*(RotArray[0]->VelRotInt);
                        accelDecel = 0.5*(RotArray[0]->AccRotInt);
                        targetPos =  2000*(RotArray[0]->PosRotInt);
                        LoadParameters(axis, velocity, accelDecel, targetPos);
                        Axes1_4 = Axes1_4 + 0x0004;
                }
                axis++;
                //Axis 3 move 1. Note lead on ball screw is .2"
                if(VertParamPos[0]){
                        velocity = 118.11*(VertArray[0]->VelVertInt);
                        accelDecel = 1.97*(VertArray[0]->AccVertInt);
                        targetPos =  7874.02*(VertArray[0]->PosVertInt);
                        LoadParameters(axis, velocity, accelDecel, targetPos);
                        Axes1_4 = Axes1_4 + 0x0008;
```

```
                }
                axis++;
                //Axis 4 move 1. Please note that the axis has a 50:1 gear box.
                if(GripParamPos[0]){
                        velocity = 8.3333*(GripperArray[0]->VelGripInt);        //Axis
has Gear box 50:1
                        accelDecel = 0.14*(GripperArray[0]->AccGripInt);
                        targetPos = 555.56*(GripperArray[0]->PosGripInt);
                        LoadParameters(axis, velocity, accelDecel, targetPos);
                        Axes1_4 = Axes1_4 + 0x0010;
                }
                axis = 1;
                motion->MoveNo1->Brush->Color=clGreen;
                motion->MoveNo2->Brush->Color=clWhite;
                motion->MoveNo3->Brush->Color=clWhite;
                motion->MoveNo4->Brush->Color=clWhite;
                //Funtion to set order of axes to move
                MoveOrder(0, Axes1_4);

                //Going to open or close gripper if selected for first path (not equal to
-1).
                if(gripoc[0] != -1){
                        switch(gripoc[0]){
                                case 0:
                                        //Closing the gripper. (picking up item)
                                        if(status == NIMC_noError){
                                                status =
flex_set_breakpoint_momo(boardID, 0, 0x00, 0x10, HOST);
                                        }
                                        //Wait for gripper to close before moving.
                                        while(!GripClose){
```

```
                                        if(status == NIMC_noError){
flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
                                        }

                                        if(status != NIMC_noError)break;

                                        GripClose = 0x002 & TriggerInputs;
                                }
                        break;
                        case 1:
                                //Opening the gripper. (placing item)
                                if(status == NIMC_noError){
                                        status =
flex_set_breakpoint_momo(boardID, 0, 0x10, 0x00, HOST);
                                }
                                //Wait for gripper to open before moving.
                                while(!GripOpen){

                                        if(status == NIMC_noError){
flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
                                        }

                                        if(status != NIMC_noError)break;

                                        GripOpen = 0x004 & TriggerInputs;
                                }
                        break;
                }
```

```
}

Axes1_4 = 0;

//Running second trajectory path if the halt button was not pressed
if(!HaltCheck){
        if(HorzParamPos[1]){
                velocity = 118.11*(HorzArray[1]->VelHorzInt);
                accelDecel = 1.97*(HorzArray[1]->AccHorzInt);
                targetPos =  7874.02*(HorzArray[1]->PosHorzInt);
                LoadParameters(axis, velocity, accelDecel, targetPos);
                Axes1_4 = Axes1_4 + 0x0002;
        }
        axis++;
        if(RotParamPos[1]){
                velocity = 30*(RotArray[1]->VelRotInt);
                accelDecel = 0.5*(RotArray[1]->AccRotInt);
                targetPos =  2000*(RotArray[1]->PosRotInt);
                LoadParameters(axis, velocity, accelDecel, targetPos);
                Axes1_4 = Axes1_4 + 0x0004;
        }
        axis++;
        if(VertParamPos[1]){
                velocity = 118.11*(VertArray[1]->VelVertInt);
                accelDecel = 1.97*(VertArray[1]->AccVertInt);
                targetPos =  7874.02*(VertArray[1]->PosVertInt);
                LoadParameters(axis, velocity, accelDecel, targetPos);
                Axes1_4 = Axes1_4 + 0x0008;
        }
        axis++;
        if(GripParamPos[1]){
```

```
                                velocity = 8.33*(GripperArray[1]->VelGripInt);
                                accelDecel = 0.14*(GripperArray[1]->AccGripInt);
                                targetPos = 555.56*(GripperArray[1]->PosGripInt);
                                LoadParameters(axis, velocity, accelDecel, targetPos);
                                Axes1_4 = Axes1_4 + 0x0010;
                        }
                        axis = 1;
                        motion->MoveNo1->Brush->Color=clWhite;
                        motion->MoveNo2->Brush->Color=clGreen;
                        motion->MoveNo3->Brush->Color=clWhite;
                        motion->MoveNo4->Brush->Color=clWhite;
                        //Funtion to set order of axes to move
                        MoveOrder(1, Axes1_4);

                        //Going to open or close gripper if selected for second path (not
equal to -1).
                        if(gripoc[1] != -1){
                                switch(gripoc[1]){
                                case 0:
                                        //Closing the gripper. (picking up item)
                                        if(status == NIMC_noError){
                                                status =
flex_set_breakpoint_momo(boardID, 0, 0x00, 0x10, HOST);
                                        }
                                        //Wait for gripper to close before moving.
                                        while(!GripClose){

                                                if(status == NIMC_noError){

flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
                                                }
```

```
                                        if(status != NIMC_noError)break;

                                        GripClose = 0x002 & TriggerInputs;
                                }
                        break;
                        case 1:
                                //Opening the gripper. (placing item)
                                if(status == NIMC_noError){
                                        status =
flex_set_breakpoint_momo(boardID, 0, 0x10, 0x00, HOST);
                                }
                                //Wait for gripper to open before moving.
                                while(!GripOpen){

                                        if(status == NIMC_noError){

flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
                                        }

                                        if(status != NIMC_noError)break;

                                        GripOpen = 0x004 & TriggerInputs;
                                }
                        break;
                        }
                }
        }

        Axes1_4 = 0;
```

```
//Running third trajectory path if the halt button was not pressed
if(!HaltCheck){
        if(HorzParamPos[2]){
                velocity = 118.11*(HorzArray[2]->VelHorzInt);
                accelDecel = 1.97*(HorzArray[2]->AccHorzInt);
                targetPos =  7874.02*(HorzArray[2]->PosHorzInt);
                LoadParameters(axis, velocity, accelDecel, targetPos);
                Axes1_4 = Axes1_4 + 0x0002;
        }
        axis++;
        if(RotParamPos[2]){
                velocity = 30*(RotArray[2]->VelRotInt);
                accelDecel = 0.5*(RotArray[2]-> AccRotInt);
                targetPos =  2000*(RotArray[2]->PosRotInt);
                LoadParameters(axis, velocity, accelDecel, targetPos);
                Axes1_4 = Axes1_4 + 0x0004;
        }
        axis++;
        if(VertParamPos[2]){
                velocity = 118.11*(VertArray[2]->VelVertInt);
                accelDecel = 1.97*(VertArray[2]->AccVertInt);
                targetPos =  7874.02*(VertArray[2]->PosVertInt);
                LoadParameters(axis, velocity, accelDecel, targetPos);
                Axes1_4 = Axes1_4 + 0x0008;
        }
        axis++;
        if(GripParamPos[2]){
                velocity = 8.33*(GripperArray[2]->VelGripInt);
                accelDecel = 0.14*(GripperArray[2]->AccGripInt);
                targetPos = 555.56*(GripperArray[2]->PosGripInt);
                LoadParameters(axis, velocity, accelDecel, targetPos);
```

```
                        Axes1_4 = Axes1_4 + 0x0010;
                }
                axis = 1;
                motion->MoveNo1->Brush->Color=clWhite;
                motion->MoveNo2->Brush->Color=clWhite;
                motion->MoveNo3->Brush->Color=clGreen;
                motion->MoveNo4->Brush->Color=clWhite;
                //Funtion to set order of axes to move
                MoveOrder(2, Axes1_4);

                //Going to open or close gripper if selected for third path (not
equal to -1).
                if(gripoc[2] != -1){
                        switch(gripoc[2]){
                        case 0:
                                //Closing the gripper. (picking up item)
                                if(status == NIMC_noError){
                                        status =
flex_set_breakpoint_momo(boardID, 0, 0x00, 0x10, HOST);
                                }
                                //Wait for gripper to close before moving.
                                while(!GripClose){

                                        if(status == NIMC_noError){

flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
                                        }

                                        if(status != NIMC_noError)break;

                                        GripClose = 0x002 & TriggerInputs;
```

```
                                        }
                                break;
                                case 1:
                                        //Opening the gripper. (placing item)
                                        if(status == NIMC_noError){
                                                status =
flex_set_breakpoint_momo(boardID, 0, 0x10, 0x00, HOST);
                                        }
                                        //Wait for gripper to open before moving.
                                        while(!GripOpen){

                                                if(status == NIMC_noError){

flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
                                                }

                                                if(status != NIMC_noError)break;

                                                GripOpen = 0x004 & TriggerInputs;
                                        }
                                break;
                                }
                        }
                }

                Axes1_4 = 0;

                //Running fourth trajectory path if the halt button was not pressed
                if(!HaltCheck){
                        if(HorzParamPos[3]){
                                velocity = 118.11*(HorzArray[3]->VelHorzInt);
```

```
        accelDecel = 1.97*(HorzArray[3]->AccHorzInt);
        targetPos =  7874.02*(HorzArray[3]->PosHorzInt);
        LoadParameters(axis, velocity, accelDecel, targetPos);
        Axes1_4 = Axes1_4 + 0x0002;
}
axis++;
if(RotParamPos[3]){
        velocity = 30*(RotArray[3]->VelRotInt);
        accelDecel = 0.5*(RotArray[3]->AccRotInt);
        targetPos =  2000*(RotArray[3]->PosRotInt);
        LoadParameters(axis, velocity, accelDecel, targetPos);
        Axes1_4 = Axes1_4 + 0x0004;
}
axis++;
if(VertParamPos[3]){
        velocity = 118.11*(VertArray[3]->VelVertInt);
        accelDecel = 1.97*(VertArray[3]->AccVertInt);
        targetPos =  7874.02*(VertArray[3]->PosVertInt);
        LoadParameters(axis, velocity, accelDecel, targetPos);
        Axes1_4 = Axes1_4 + 0x0008;
}
axis++;
if(GripParamPos[3]){
        velocity = 87.33*(GripperArray[3]->VelGripInt);
        accelDecel = 0.14*(GripperArray[3]->AccGripInt);
        targetPos = 555.56*(GripperArray[3]->PosGripInt);
        LoadParameters(axis, velocity, accelDecel, targetPos);
        Axes1_4 = Axes1_4 + 0x0010;
}
axis = 1;
motion->MoveNo1->Brush->Color=clWhite;
```

```
                        motion->MoveNo2->Brush->Color=clWhite;
                        motion->MoveNo3->Brush->Color=clWhite;
                        motion->MoveNo4->Brush->Color=clGreen;
                        //Funtion to set order of axes to move
                        MoveOrder(3, Axes1_4);

                        //Going to open or close gripper if selected for third path (not
equal to -1).
                        if(gripoc[3] != -1){
                                switch(gripoc[3]){
                                case 0:
                                        //Closing the gripper. (picking up item)
                                        if(status == NIMC_noError){
                                                status =
flex_set_breakpoint_momo(boardID, 0, 0x00, 0x10, HOST);
                                        }
                                        //Wait for gripper to close before moving.
                                        while(!GripClose){

                                                if(status == NIMC_noError){

flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
                                                }
                                                if(status != NIMC_noError)break;
                                                GripClose = 0x002 & TriggerInputs;
                                        }
                                break;
                                case 1:
                                        //Opening the gripper. (placing item)
                                        if(status == NIMC_noError){
```

```
                                                status =
flex_set_breakpoint_momo(boardID, 0, 0x10, 0x00, HOST);
                                        }
                                        //Wait for gripper to open before moving.
                                        while(!GripOpen){

                                                if(status == NIMC_noError){
flex_read_hs_cap_status_rtn(boardID, 0, &TriggerInputs);
                                                }
                                                if(status != NIMC_noError)break;
                                                GripOpen = 0x004 & TriggerInputs;
                                        }
                                break;
                                }
                        }
                }

                Axes1_4 = 0;

                //Repeating moves if one of repeat check boxes is checked
                while((RepeatHorz || RepeatRot || RepeatVert || RepeatGrip) &&
!HaltCheck){
                        axis = 1;
                        for(int i=0;i<4;i++){
                                Axes1_4 = 0;
                                if(RepeatHorz && HorzParamPos[i]){
                                        velocity = 118.11*(HorzArray[i]->VelHorzInt);
                                        accelDecel = 1.97*(HorzArray[i]->AccHorzInt);
                                        targetPos = 7874.02*(HorzArray[i]->PosHorzInt);
                                        LoadParameters(axis, velocity, accelDecel,
targetPos);
```

```
                        Axes1_4 = Axes1_4 + 0x0002;
                        BrkPon = BrkPon + 0x02;     //Turning on axis 2
motion led
                }
                axis++;
                if(RepeatRot && RotParamPos[i]){
                        velocity = 30*(RotArray[i]->VelRotInt);
                        accelDecel = 0.5*(RotArray[i]->AccRotInt);
                        targetPos =  2000*(RotArray[i]->PosRotInt);
                        LoadParameters(axis, velocity, accelDecel,
targetPos);
                        Axes1_4 = Axes1_4 + 0x0004;
                        BrkPon = BrkPon + 0x04;     //Turning on axis 2
motion led
                }
                axis++;
                if(RepeatVert && VertParamPos[i]){
                        velocity = 118.11*(VertArray[i]->VelVertInt);
                        accelDecel = 1.97*(VertArray[i]->AccVertInt);
                        targetPos =  7874.02*(VertArray[i]->PosVertInt);
                        LoadParameters(axis, velocity, accelDecel,
targetPos);
                        Axes1_4 = Axes1_4 + 0x0008;
                        BrkPon = BrkPon + 0x08;     //Turning on axis 2
motion led
                }
                axis++;
                if(RepeatGrip && GripParamPos[i]){
                        velocity = 8.33*(GripperArray[i]->VelGripInt);
                        accelDecel = 1.97*(GripperArray[i]->AccGripInt);
                        targetPos = 555.56*(GripperArray[i]->PosGripInt);
```

```
                LoadParameters(axis, velocity, accelDecel,
targetPos);
                Axes1_4 = Axes1_4 + 0x0010;
        }
        axis=1;
        //Setting move #1 to green
        if(i==0){
                motion->MoveNo1->Brush->Color=clGreen;
                motion->MoveNo2->Brush->Color=clWhite;
                motion->MoveNo3->Brush->Color=clWhite;
                motion->MoveNo4->Brush->Color=clWhite;
        }
        //Setting move #2 to green
        if(i==1){
                motion->MoveNo1->Brush->Color=clWhite;
                motion->MoveNo2->Brush->Color=clGreen;
                motion->MoveNo3->Brush->Color=clWhite;
                motion->MoveNo4->Brush->Color=clWhite;
        }
        //Setting move #3 to green
        if(i==2){
                motion->MoveNo1->Brush->Color=clWhite;
                motion->MoveNo2->Brush->Color=clWhite;
                motion->MoveNo3->Brush->Color=clGreen;
                motion->MoveNo4->Brush->Color=clWhite;
        }
        //Setting move #4 to green
        if(i==3){
                motion->MoveNo1->Brush->Color=clWhite;
                motion->MoveNo2->Brush->Color=clWhite;
                motion->MoveNo3->Brush->Color=clWhite;
```

```
                        motion->MoveNo4->Brush->Color=clGreen;
                }
                //Funtion to set order of axes to move
                MoveOrder(i, Axes1_4);

                //Going to open or close gripper if selected for the ith
path (not equal to -1).
                //Checking to see if gripper is closed already
                if(gripoc[i] != -1){

                        switch(gripoc[i]){
                                case 0:
                                //Closing the gripper. (picking up item)
                                        if(status == NIMC_noError){
                                                status =
flex_set_breakpoint_momo(boardID, 0, 0x00, 0x10, HOST);
                                        }
                                        break;
                                case 1:
                                //Opening the gripper. (placing item)
                                        if(status == NIMC_noError){
                                                status =
flex_set_breakpoint_momo(boardID, 0, 0x10, 0x00, HOST);
                                        }
                                        break;
                                }
                        }

                }

                if(status != NIMC_noError)break;
```

```
                        }

                        AxisStatus->Close->Enabled = true;
                        Suspend();
        }

}
//---------------------------------------------------------------------------
void MoveThread::MoveOrder(int AxisNum, long AxesToMove){

        bool MoveAxesCheck = false;

        for(int i=0; i<4; i++){
                //Moving each axis according to the order selected
                //0=first move; 1=second move; 2=third move; 3=fourth move
                MoveAxesOrder = 0;
                BrkPon = 0;
                MoveAxesCheck = false;
                switch(i){
                        //First move
                        case 0:
                                if(HorzArray[AxisNum]->OrderHorz==0){
                                        MoveAxesOrder = 0x0002;
                                        BrkPon = BrkPon + 0x02;     //Turning on axis 1
motion led
                                        MoveAxesCheck = true;
                                }
                                if(RotArray[AxisNum]->OrderRot==0){
                                        MoveAxesOrder = MoveAxesOrder + 0x0004;
                                        BrkPon = BrkPon + 0x04;     //Turning on axis 2
motion led
```

```
                                        MoveAxesCheck = true;
                                }
                                if(VertArray[AxisNum]->OrderVert==0){
                                        MoveAxesOrder = MoveAxesOrder + 0x0008;
                                        BrkPon = BrkPon + 0x08;     //Turning on axis 3
motion led.
                                        MoveAxesCheck = true;
                                }
                                if(GripperArray[AxisNum]->OrderGrip==0){
                                        MoveAxesOrder = MoveAxesOrder + 0x0010;
                                        MoveAxesCheck = true;
                                }
                        break;
                        //Second move
                        case 1:
                                if(HorzArray[AxisNum]->OrderHorz==1){
                                        MoveAxesOrder = 0x0002;
                                        BrkPon = BrkPon + 0x02;     //Turning on axis 1
motion led
                                        MoveAxesCheck = true;
                                }
                                if(RotArray[AxisNum]->OrderRot==1){
                                        MoveAxesOrder = MoveAxesOrder + 0x0004;
                                        BrkPon = BrkPon + 0x04;     //Turning on axis 2
motion led
                                        MoveAxesCheck = true;
                                }
                                if(VertArray[AxisNum]->OrderVert==1){
                                        MoveAxesOrder = MoveAxesOrder + 0x0008;
                                        BrkPon = BrkPon + 0x08;     //Turning on axis 3
motion led.
```

```
                                MoveAxesCheck = true;
                        }
                        if(GripperArray[AxisNum]->OrderGrip==1){
                                MoveAxesOrder = MoveAxesOrder + 0x0010;
                                MoveAxesCheck = true;
                        }
                break;
                //Third move
                case 2:
                        if(HorzArray[AxisNum]->OrderHorz==2){
                                MoveAxesOrder = 0x0002;
                                BrkPon = BrkPon + 0x02;     //Turning on axis 1
motion led
                                MoveAxesCheck = true;
                        }
                        if(RotArray[AxisNum]->OrderRot==2){
                                MoveAxesOrder = MoveAxesOrder + 0x0004;
                                BrkPon = BrkPon + 0x04;     //Turning on axis 2
motion led
                                MoveAxesCheck = true;
                        }
                        if(VertArray[AxisNum]->OrderVert==2){
                                MoveAxesOrder = MoveAxesOrder + 0x0008;
                                BrkPon = BrkPon + 0x08;     //Turning on axis 3
motion led.
                                MoveAxesCheck = true;
                        }
                        if(GripperArray[AxisNum]->OrderGrip==2){
                                MoveAxesOrder = MoveAxesOrder + 0x0010;
                                MoveAxesCheck = true;
                        }
```

```
                        break;
                        //Fourth and final move
                        case 3:
                                if(HorzArray[AxisNum]->OrderHorz==3){
                                        MoveAxesOrder = 0x0002;
                                        BrkPon = BrkPon + 0x02;     //Turning on axis 1
motion led
                                        MoveAxesCheck = true;
                                }
                                if(RotArray[AxisNum]->OrderRot==3){
                                        MoveAxesOrder = MoveAxesOrder + 0x0004;
                                        BrkPon = BrkPon + 0x04;     //Turning on axis 2
motion led
                                        MoveAxesCheck = true;
                                }
                                if(VertArray[AxisNum]->OrderVert==3){
                                        MoveAxesOrder = MoveAxesOrder + 0x0008;
                                        BrkPon = BrkPon + 0x08;     //Turning on axis 3
motion led.
                                        MoveAxesCheck = true;
                                }
                                if(GripperArray[AxisNum]->OrderGrip==3){
                                        MoveAxesOrder = MoveAxesOrder + 0x0010;
                                        MoveAxesCheck = true;
                                }
                        break;
                }
                        //Checking to make sure that an axis has been commanded to move.
                if(MoveAxesCheck){
                        //Mask bit to move correct axes
                        MoveAxesOrder = AxesToMove & MoveAxesOrder;
```

```
                        //Move axes
                        MoveAxes(BrkPon);
                }
        }
}

//---------------------------------------------------------------------------
void __fastcall MoveThread::DisplayASUpdate()
{
        u8 FLS;
        u8 RLS;
        u16 HS;
        u16 GripStatus;

        for (int i=1;i<5;i++){
                if(status == NIMC_noError){
                        status = flex_read_pos_rtn(boardID, i, &positionRet[i-1]);
                }
                if(status == NIMC_noError){
                        status = flex_read_velocity_rtn(boardID, i, &velocityRet[i-1]);
                }
        }

        if(status == NIMC_noError){
                status = flex_read_limit_status_rtn(boardID, NIMC_LIMIT_INPUTS, &FLS,
&RLS);
        }
        if(status == NIMC_noError){
                status = flex_read_home_input_status_rtn(boardID, &HS);
        }
        if(status == NIMC_noError){
```

```
        status = flex_read_hs_cap_status_rtn(boardID, 0, &GripStatus);
}
//Checking horizontal axis switches
if((FLS & 0x0002)){
        AxisStatus->FLSActiveHorz->Brush->Color=clRed;
}else{AxisStatus->FLSActiveHorz->Brush->Color=clWhite;}

if((RLS & 0x0002)){
        AxisStatus->RLSActiveHorz->Brush->Color=clRed;
}else{AxisStatus->RLSActiveHorz->Brush->Color=clWhite;}

if((HS & 0x0002)){
        AxisStatus->HSActiveHorz->Brush->Color=clRed;
}else{AxisStatus->HSActiveHorz->Brush->Color=clWhite;}
//Checking rotational axis switch
if((HS & 0x0004)){
        AxisStatus->HSActiveRot->Brush->Color=clRed;
}else{AxisStatus->HSActiveRot->Brush->Color=clWhite;}
//Checking vertical axis switches
if((FLS & 0x0008)){
        AxisStatus->FLSActiveVert->Brush->Color=clRed;
}else{AxisStatus->FLSActiveVert->Brush->Color=clWhite;}

if((RLS & 0x0008)){
        AxisStatus->RLSActiveVert->Brush->Color=clRed;
}else{AxisStatus->RLSActiveVert->Brush->Color=clWhite;}

if((HS & 0x0008)){
        AxisStatus->HSActiveVert->Brush->Color=clRed;
}else{AxisStatus->HSActiveVert->Brush->Color=clWhite;}
//Checking gripper axis switches
```

```
        if((GripStatus & 0x0002)){
                AxisStatus->GripperClosedLt->Brush->Color=clRed;
                AxisStatus->GripperOpenLt->Brush->Color=clWhite;
        }else{
                AxisStatus->GripperClosedLt->Brush->Color=clWhite;
                AxisStatus->GripperOpenLt->Brush->Color=clRed;
        }
        if((HS & 0x0010)){
                 AxisStatus->HSActiveGrip->Brush->Color=clRed;
        }else{AxisStatus->HSActiveGrip->Brush->Color=clWhite;}


        AxisStatus->VelHorzFB-
>Text=FloatToStrF((float)(velocityRet[0]/118.11),ffFixed,2,1);
        AxisStatus->PosHorzFB-
>Text=FloatToStrF((float)(positionRet[0]/7874.02),ffFixed,2,2);
        AxisStatus->VelRotFB->Text=FloatToStrF((float)(velocityRet[1]/30),ffFixed,1,1);
        AxisStatus->PosRotFB->Text=FloatToStrF((float)(positionRet[1]/2000),ffFixed,2,2);
        AxisStatus->VelVertFB-
>Text=FloatToStrF((float)(velocityRet[2]/118.11),ffFixed,2,1);
        AxisStatus->PosVertFB-
>Text=FloatToStrF((float)(positionRet[2]/7874.02),ffFixed,2,2);
        AxisStatus->VelGripFB-
>Text=FloatToStrF((float)(velocityRet[3]/8.33),ffFixed,2,1);
        AxisStatus->PosGripFB-
>Text=FloatToStrF((float)(positionRet[3]/555.55),ffFixed,2,2);

        AxisStatus->MotionBar->SimpleText="In Motion.....";

        AxisStatus->Refresh();
```

```
}
//---------------------------------------------------------------------------
void MoveThread::MoveAxes(int mBrkPon)
{

        status = flex_read_csr_rtn(boardID, &csr);

        //Turning on motion leds for axes that are commanded to move.
        if(status == NIMC_noError){
                status = flex_set_breakpoint_momo(boardID, 0, BrkPon, BrkPoff, HOST);
        }

        if(!HaltCheck){

                if(status == NIMC_noError){
                        status = flex_start(boardID, 0, MoveAxesOrder);
                }
                //Report Non-modal error.
                if(status){
                        ErrorHandler(status,0,0);
                }
        }

        do{
                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 1, &AxisRtn1);
                }

                if(status == NIMC_noError){
                        status = flex_read_axis_status_rtn(boardID, 2, &AxisRtn2);
                }
```

```
        if(status == NIMC_noError){
                status = flex_read_axis_status_rtn(boardID, 3, &AxisRtn3);
        }

        if(status == NIMC_noError){
                status = flex_read_axis_status_rtn(boardID, 4, &AxisRtn4);
        }

        Synchronize(DisplayASUpdate);

        motion->CheckModalError();

        //If any function to the FlexMotion board failed due to a
        //non-modal error exit the loop.
        if(status){
                ErrorHandler(status, 0, 0);
                break;
        }
        if(HaltCheck){
                break;
        }
        //Test against the move complete bit or the axis off bit
}while(!((AxisRtn1 & (NIMC_MOVE_COMPLETE_BIT | NIMC_AXIS_OFF_BIT)) &&
        (AxisRtn2 & (NIMC_MOVE_COMPLETE_BIT | NIMC_AXIS_OFF_BIT)) &&
        (AxisRtn3 & (NIMC_MOVE_COMPLETE_BIT | NIMC_AXIS_OFF_BIT)) &&
        (AxisRtn4 & (NIMC_MOVE_COMPLETE_BIT | NIMC_AXIS_OFF_BIT))));

Synchronize(DisplayASUpdate);

AxisStatus->MotionBar->SimpleText="Move Complete.";
```

```
        BrkPon = 0x00;

        BrkPoff = 0x0E;

        //Turning off motion leds for axes that are commanded to move.
        if(status == NIMC_noError){
                status = flex_set_breakpoint_momo(boardID, 0, BrkPon, BrkPoff, HOST);
        }
        BrkPoff = 0x00;

        AxisStatus->Refresh();
}
//---------------------------------------------------------------------------
```

Calibration Menu Code (Calibration.cpp)

```
//---------------------------------------------------------------------------
#include <vcl.h>
#include <graphics.hpp>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
#include <condefs.h>
#include "Calibration.h"
#include "inmotion.h"
#include "axesparam.h"
#include "GifLZW.hpp"
#include "TIFLZW.hpp"
#include "ImageEnIO.hpp"
#define HOST 0xFF
//---------------------------------------------------------------------------
#pragma hdrstop
#pragma package(smart_init)
#pragma link "ImageEnIO"
#pragma link "ImageEnProc"
#pragma link "ImageEnView"
#pragma link "VideoCap"
#pragma link "ImageEnIO"
#pragma link "ImageEnProc"
#pragma link "ImageEnView"
#pragma link "VideoCap"
#pragma link "IEOpenSaveDlg"
#pragma link "IEOpenSaveDlg"
#pragma link "ImageEnIO"
#pragma link "ImageEnProc"
```

```
#pragma link "ImageEnView"
#pragma link "VideoCap"
#pragma link "IEVect"
#pragma link "ieview"
#pragma package(smart_init)
#pragma link "IEVect"
#pragma link "ieview"
#pragma link "ImageEnView"
#pragma link "VideoCap"
#pragma link "ImageEn"
#pragma resource "*.dfm"
//---------------------------------------------------------------------------
//Function prototypes written in calibration.cpp
//Black fill function
void BlackFill(int m, int n);
//Function to get raw data from image and store in containers.
void GetRawData(Graphics::TBitmap *Image);
//Function to find far left x
int FarLeftX(int m, int n);
//Function to find far right x
int FarRightX(int m, int n);
//Function to firtnd last y
int BottomY(int m, int n);
//Function to find and check for ccrrect color
bool ColorFind(int LowerBnd[3], int UpperBnd[3], int Color);
//Average 3x3 area of pixels
int Average(BYTE Image[][74], int x, int y)
{
        int i, j, sum=0;

        for (j=-1; j<=1; j++)
```

```
                for (i=-1; i<=1; i++)
                      sum += Image[x+i][y+j];

        return INT(sum/9.0);
}
//-------------------------------------------------------------------------
//Function Prototypes written in inmotion.cpp
//Errorhandler will print the error discription to screen
void ErrorHandler(i32 errorCode, u16 commandID, u16 resourceID);
//Function to load the parameters to each axis
void LoadParameters(u8 mAxis, f64 mAcc, f64 mVel, i32 mPos);
//-------------------------------------------------------------------------
//Variable declarations
i32 position;                   //Target Position in counts
f64 accdec;                      //Acceleration value in rev/s^2
f64 vel;                         //Acceleration in rpm
int PixelValue;                  //Variable to store pixel value;
int ColorSelected=-1;            //Variable to hold the color selected from the combobox
int FarLeftx = 0;                //Variable for far left x pixel location
int FarRightx = 0;               //Variable for far right x pixel location
int Beginy = 0;                  //Variable to start from for the y bottom location
int Bottomy = 0;                 //Variable for last y pixel location
int Middley = 0;                 //Variable to hold middle of connector y
int Middlex = 0;                 //Variable to hold middle of connecto x
int Firstx = 0;                  //Variable to hold first white pixel along x
int Firsty = 0;                  //Variable to hold first white pixel along y
int Pixelx = 0;                  //Pixel value saved to store desired postion along x
int Pixely = 0;                  //Pixel value saved to store desired postion along y
int PixelxDes = 0;               //Desired insulator location along x
int PixelyDes = 0;               //Desired insulator location along y
int ColorLowerBound[3];          //Array to hold lower threshold for RGB (R=0, G=1, B=2);
```

```
int ColorUpperBound[3];          //Array to hold upper threshold for RGB (R=0, G=1, B=2);
long ImageHandle;                //Image handle for VideoOCX
long ImageData;                  //Address to image data.
float HorzOffset = 0;            //Distance of travel for horz axis when picking up cap
float VertOffset1 = 0;           //Distance of travel point 1 for vert axis when picking
up cap
float VertOffset2 = 0;           //Distance of travel point 2 for vert axis when picking
up cap
bool CamCheck = false;           //Check variable to make sure the camera function worked.
bool CameraOn = false;           //Check variable to make sure the camera is working.
bool PicInClip = false;          //Check variable to make sure picture is in clipboard.
bool CrossHair = false;          //Check variable to draw crosshairs
bool ImageCapture = false;       //Check variable to make sure image has loaded
bool CalMenuCheck = false;       //Check variable to make sure calibration has been
performed
BYTE PixelRGB[3];                //Array to hold each individual pixel byte.
BYTE *ImageRawData;              //Pointer to pixel datas
BYTE ImageDataMatrixRed[74][74];        //Raw data for red primary
BYTE ImageDataMatrixBlue[74][74];       //Raw data for blue primary
BYTE ImageDataMatrixGreen[74][74];      //Raw data for green primary
BYTE ImageDataBW[74][74];               //Raw data storage for Black and white image
BYTE ImageBlkFill[74][74];              //Raw data for black fill storage

//Variable to draw a cross on the selected color insulator
Windows::TPoint points[6];

//Dynamically allocating memory from the heap for CapPicture
Graphics::TBitmap *CapPicture=new Graphics::TBitmap();

//External variables from other sources
extern bool InitCheck;
```

```
extern u8 boardID;
extern u8 status;
extern u16 csr;
extern u8 axis;
extern bool HorzParamPos[4];
extern bool VertParamPos[4];
extern HorizontalAxis *HorzArray[4];
extern VerticalAxis *VertArray[4];
extern RotationAxis *RotArray[4];
extern GripperAxis *GripperArray[4];
extern int gripoc[4];
extern i32 positionRet[4];

TCalibrateMenu *CalibrateMenu;

TRect FrameCap;         //Creating rectangular frame for cap location

//---------------------------------------------------------------------------

__fastcall TCalibrateMenu::TCalibrateMenu(TComponent* Owner)
        : TForm(Owner)
{
        VideoPanel->Visible = false;
        ColorSelPanel->Visible = false;
        //Drawing circle on ImageCircle canvas
        FrameCap.Left=123; FrameCap.Right=197;
        FrameCap.Top=83;  FrameCap.Bottom=157;
        ImageCirclePanel->Visible = true;
        ImageCircle->Canvas->Pen->Color=clYellow;
        ImageCircle->Canvas->Pen->Mode=pmCopy;
        ImageCircle->Canvas->Pen->Width=2;
```

```
        ImageCircle->Canvas->Brush->Style=bsClear;
        ImageCircle->Canvas-
>Ellipse(FrameCap.Left,FrameCap.Top,FrameCap.Right,FrameCap.Bottom);
}
//---------------------------------------------------------------------------

void __fastcall TCalibrateMenu::ExitButtonClick(TObject *Sender)
{
        if(CameraOn){
                //Stop video
                CamCheck = VideoOCXCal->Close();
                CameraOn = false;

                if(CamCheck){
                        Application->MessageBox("Video capture device turned
off.","Message",MB_OK);
                }
                else{
                        Application->MessageBox("Video capture device failed to turn
off.","Error",MB_OK);
                }
        }
        if(ImageCapture){
                delete CapPicture;
        }
        //Setting operation mode back to absolute
        for(axis=1;axis<5;axis++){
                if(status == NIMC_noError){
                        status = flex_set_op_mode(boardID,axis,NIMC_ABSOLUTE_POSITION);
                }
        }
```

```
        ImageCapture = false;
        CalibrateMenu->Visible=false;
}
//---------------------------------------------------------------------------

void __fastcall TCalibrateMenu::ViewVideoButClick(TObject *Sender)
{
        ImageCirclePanel->Visible = false;

        ColorSelPanel->Visible = false;

        VideoPanel->Visible = true;

        VideoOCXCal->SetResolution(320,240);

        if(!CameraOn){
                //Initialize video camera
                CamCheck = VideoOCXCal->Init();
                //View live video
                if(CamCheck){
                        CamCheck = VideoOCXCal->SetPreview(true);
                        CameraOn = true;
                }
                else{
                        Application->MessageBoxA("Video Capture device is in use or has
not initialized.","Error",MB_OK);
                }
        }
        else{
                Application->MessageBoxA("Video Capture device is already
on.","Error",MB_OK);
```

```
        }
}
//---------------------------------------------------------------------------

void __fastcall TCalibrateMenu::OpenButtonClick(TObject *Sender)
{
        OpenDialog1->Filter = "Bmp files (*.bmp)|*.BMP";
        if (OpenDialog1->Execute())
        {
                ImageCirclePanel->Visible = false;
                VideoPanel->Visible = false;
                ColorSelPanel->Visible = true;
                ColorSelImage->Picture->Bitmap->LoadFromFile(OpenDialog1->FileName);
        }
}
//---------------------------------------------------------------------------

void __fastcall TCalibrateMenu::SavButtonClick(TObject *Sender)
{
        //Setting the first move position parameters
        HorzArray[0]->PosHorzInt = HorzOffset;
        HorzArray[0]->OrderHorz = 0;            //Setting horz axis move #1
        VertArray[0]->PosVertInt = VertOffset1;
        VertArray[0]->OrderVert = 1;            //Setting vert axis move #2
        gripoc[0] = 0;                          //Closing gripper

        //Setting the second move position parameters
        VertArray[1]->PosVertInt = VertOffset2;
        //VertArray[1]->VelVertInt = 1;           //Putting a default value in
        //VertArray[1]->AccVertInt = 1;           //Putting a default value in
        VertArray[1]->OrderVert = 0;            //Setting vert axis move #1
```

```
GripperArray[1]->PosGripInt = 0;
//GripperArray[1]->VelGripInt = 1;
//GripperArray[1]->AccGripInt = 1;
GripperArray[1]->OrderGrip = 1;         //Setting the gripper axis move #2

//Setting the third move position paramters
VertArray[2]->PosVertInt = VertOffset1;
//VertArray[2]->VelVertInt = 1;           //Putting a default value in
//VertArray[2]->AccVertInt = 1;           //Putting a default value in
VertArray[2]->OrderVert = 0;            //Setting vert axis move #1
gripoc[2] = 1;                          //opening gripper

//Setting the fouth move position paramters.  Sending back home.
VertArray[3]->PosVertInt = 0;
//VertArray[3]->VelVertInt = 1;           //Putting a default value in
//VertArray[3]->AccVertInt = 1;           //Putting a default value in
VertArray[3]->OrderVert = 0;            //Setting vert axis move #1
HorzArray[3]->PosHorzInt = 0;
//HorzArray[3]->VelHorzInt = 1;           //Putting a default value in
//HorzArray[3]->AccHorzInt = 1;           //Putting a default value in
HorzArray[3]->OrderHorz = 1;            //Setting horz axis move #2
GripperArray[3]->PosGripInt = 0;
//GripperArray[3]->VelGripInt = 1;
//GripperArray[3]->AccGripInt = 1;
GripperArray[3]->OrderGrip = 1;         //Setting gripper axis move #2

//Showing first parameter move in text boxes
motion->PosHorz->Text = HorzArray[0]->PosHorzInt;
motion->HorzOrdRG->ItemIndex = HorzArray[0]->OrderHorz;
motion->PosVert->Text = VertArray[0]->PosVertInt;
motion->VertOrdRG->ItemIndex = VertArray[0]->OrderVert;
```

```
        motion->GripOC->ItemIndex = gripoc[0];

        PixelxDes = Pixelx;                         //Saving desired pixel location x
direction
        PixelyDes = Pixely;                         //Saving desired pixel location y
direction

        CalMenuCheck = true;
}
//---------------------------------------------------------------------------

void __fastcall TCalibrateMenu::PageControlChange(TObject *Sender)
{
        ImageCirclePanel->Visible = false;

        ColorSelPanel->Visible = false;

        VideoPanel->Visible = true;

        VideoOCXCal->SetResolution(320,240);

        if(!CameraOn){
                //Initialize video camera
                CamCheck = VideoOCXCal->Init();
                //View live video
                if(CamCheck){
                        CamCheck = VideoOCXCal->SetPreview(true);
                        CameraOn = true;
                }
                else{
```

```
                        Application->MessageBoxA("Video Capture device is in use or has
not initialized.","Error",MB_OK);
                }
        }
}
//---------------------------------------------------------------------------
//Page control Align tab
//---------------------------------------------------------------------------
//Will move vertical axis up until mouse button is released.
void __fastcall TCalibrateMenu::UpButtonMouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 3;

        position = 0;
        accdec = 5;
        vel = 200;

        if(InitCheck){

                status = flex_read_csr_rtn(boardID, &csr);

                if(status == NIMC_noError){
                        status = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
                }

                LoadParameters(axis, vel, accdec, position);

                if(status == NIMC_noError){
                        status = flex_start(boardID, axis, axis);
                }
```

```
                //Report Non-modal error.
                if(status){
                        ErrorHandler(status,0,0);
                        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
                }
        }
        else{
                Application->MessageBox("Controller must be initialized before motion can
occur.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------
//Stop moving vertical axis up.
void __fastcall TCalibrateMenu::UpButtonMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 3;

        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
}
//---------------------------------------------------------------------------
//Will move vertical axis down until mouse button is released.
void __fastcall TCalibrateMenu::DownButtonMouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 3;

        position = 0;
        accdec = 5;
        vel = -200;
```

```
        if(InitCheck){

                status = flex_read_csr_rtn(boardID, &csr);

                if(status == NIMC_noError){
                        status = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
                }

                LoadParameters(axis, vel, accdec, position);

                if(status == NIMC_noError){
                        status = flex_start(boardID, axis, axis);
                }
                //Report Non-modal error.
                if(status){
                        ErrorHandler(status,0,0);
                        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
                }
        }
        else{
                Application->MessageBox("Controller must be initialized before motion can
occur.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------
//Stop moving vertical axis down.
void __fastcall TCalibrateMenu::DownButtonMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 3;
```

```
        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
}
//---------------------------------------------------------------------------
//Will move horizontal axis left until mouse button is released.
void __fastcall TCalibrateMenu::LeftButtonMouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 1;

        position = 0;
        accdec = 5;
        vel = 200;

        if(InitCheck){

                status = flex_read_csr_rtn(boardID, &csr);

                if(status == NIMC_noError){
                        status = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
                }

                LoadParameters(axis, vel, accdec, position);

                if(status == NIMC_noError){
                        status = flex_start(boardID, axis, axis);
                }
                //Report Non-modal error.
                if(status){
                        ErrorHandler(status,0,0);
                        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
                }
```

```
        }
        else{
                Application->MessageBox("Controller must be initialized before motion can
occur.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------
//Stop moving horizontal axis left.
void __fastcall TCalibrateMenu::LeftButtonMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 1;

        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
}
//---------------------------------------------------------------------------
//Will move horizontal axis right until mouse button is released.
void __fastcall TCalibrateMenu::RightButtonMouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 1;

        position = 0;
        accdec = 5;
        vel = -200;

        if(InitCheck){

                status = flex_read_csr_rtn(boardID, &csr);

                if(status == NIMC_noError){
```

```
                        status = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
                }

                LoadParameters(axis, vel, accdec, position);

                if(status == NIMC_noError){
                        status = flex_start(boardID, axis, axis);
                }
                //Report Non-modal error.
                if(status){
                        ErrorHandler(status,0,0);
                        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
                }
        }
        else{
                Application->MessageBox("Controller must be initialized before motion can
occur.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------
//Stop moving horizontal axis right.
void __fastcall TCalibrateMenu::RightButtonMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 1;

        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
}
//---------------------------------------------------------------------------
//Will move rotational axis counter clockwise until mouse button is released
void __fastcall TCalibrateMenu::RotateCCWButtonMouseDown(TObject *Sender,
```

```
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 2;

        position = 0;
        accdec = 5;
        vel = -200;

        if(InitCheck){

                status = flex_read_csr_rtn(boardID, &csr);

                if(status == NIMC_noError){
                        status = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
                }

                LoadParameters(axis, vel, accdec, position);

                if(status == NIMC_noError){
                        status = flex_start(boardID, axis, axis);
                }
                //Report Non-modal error.
                if(status){
                        ErrorHandler(status,0,0);
                        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
                }
        }
        else{
                Application->MessageBox("Controller must be initialized before motion can
occur.","Error",MB_OK);
        }
```

```
}
//---------------------------------------------------------------------------
//Stop moving horizontal axis counter clockwise
void __fastcall TCalibrateMenu::RotateCCWButtonMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 2;

        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
}
//---------------------------------------------------------------------------
//Will move rotational axis clockwise untill mouse button is released
void __fastcall TCalibrateMenu::RotateCWButtonMouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 2;

        position = 0;
        accdec = 5;
        vel = 200;

        if(InitCheck){

                status = flex_read_csr_rtn(boardID, &csr);

                if(status == NIMC_noError){
                        status = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
                }

                LoadParameters(axis, vel, accdec, position);
```

```
                if(status == NIMC_noError){
                        status = flex_start(boardID, axis, axis);
                }
                //Report Non-modal error.
                if(status){
                        ErrorHandler(status,0,0);
                        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
                }
        }
        else{
                Application->MessageBox("Controller must be initialized before motion can
occur.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------
//Stop moving rotation axis clockwise
void __fastcall TCalibrateMenu::RotateCWButtonMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        axis = 2;

        status = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, axis);
}
//---------------------------------------------------------------------------
//Preview capacitor position in circle
void __fastcall TCalibrateMenu::PrevButtonClick(TObject *Sender)
{
        VideoPanel->Visible = false;

        ColorSelPanel->Visible = false;
```

```
        ImageCirclePanel->Visible = true;

        CamCheck = VideoOCXCal->CaptureToClipboard();

        if(CamCheck){
                //Drawing circle outline for capicitor line up.
                ImageCircle->Picture->Assign(Clipboard());
                ImageCircle->Canvas->Pen->Color=clYellow;
                ImageCircle->Canvas->Pen->Mode=pmCopy;
                ImageCircle->Canvas->Pen->Width=2;
                ImageCircle->Canvas->Brush->Style=bsClear;
                ImageCircle->Canvas-
>Ellipse(FrameCap.Left,FrameCap.Top,FrameCap.Right,FrameCap.Bottom);

        }
}
//---------------------------------------------------------------------------
//Record the position of horizontal and vertical axis for capacitor pick up.
void __fastcall TCalibrateMenu::SetBut1Click(TObject *Sender)
{
        for (int i=1;i<5;i++){
                if(status == NIMC_noError){
                        status = flex_read_pos_rtn(boardID, i, &positionRet[i-1]);
                }
        }

        HorzOffset = positionRet[0]/7874.02;
        VertOffset1 = positionRet[2]/7874.02;
        SetBut2->Enabled = true;
}
//---------------------------------------------------------------------------
```

```
//Record the position of the horiz and vert axis when cap is in camera view
void __fastcall TCalibrateMenu::SetBut2Click(TObject *Sender)
{

        for (int i=1;i<5;i++){
                if(status == NIMC_noError){
                        status = flex_read_pos_rtn(boardID, i, &positionRet[i-1]);
                }
        }

        HorzOffset = HorzOffset - positionRet[0]/7874.02;
        VertOffset1 = VertOffset1 - positionRet[2]/7874.02;
        VertOffset2 = VertOffset1 + 2;

        //Reseting position to zero.
        for(axis=1;axis<5;axis++){
                if(status == NIMC_noError){
                        status = flex_reset_pos(boardID,axis,0,0,HOST);
                }
        }
}
//---------------------------------------------------------------------------
void __fastcall TCalibrateMenu::CameraViewButClick(TObject *Sender)
{
        VideoPanel->Visible = true;

        ColorSelPanel->Visible = false;

        ImageCirclePanel->Visible = false;
}
//---------------------------------------------------------------------------
```

```
//Page control Color tab
//---------------------------------------------------------------------------
//Selecting the color with the mouse
void __fastcall TCalibrateMenu::ColorSelImageMouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
        XPosition->Text = X;
        YPosition->Text = Y;

        ImageRawData=(BYTE *)CapPicture->ScanLine[Y];

        PixelRGB[0] = (int)ImageRawData[3*X+2];           //RedValue
        PixelRGB[1] = (int)ImageRawData[3*X+1];         //Green Value
        PixelRGB[2] = (int)ImageRawData[3*X];         //Blue Value

        PixelValue = ColorSelImage->Canvas->Pixels[X][Y];

        RedEdit->Text = PixelRGB[0];
        GreenEdit->Text = PixelRGB[1];
        BlueEdit->Text = PixelRGB[2];
}
//---------------------------------------------------------------------------
void __fastcall TCalibrateMenu::ColorComboBoxChange(TObject *Sender)
{
        bool ColorFound = false;
        ColorSelected = ColorComboBox->ItemIndex;       //0=Blue; 1=Black; 2=White;
3=Green

        if(ImageCapture){

                switch(ColorSelected){
```

```
                        //Blue has been selected
                        case 0:
                                ColorLowerBound[0] = 55;
                                ColorUpperBound[0] = 113;
                                ColorLowerBound[1] = 81;
                                ColorUpperBound[1] = 137;
                                ColorLowerBound[2] = 120;
                                ColorUpperBound[2] = 180;

                                ColorFound = ColorFind(ColorLowerBound, ColorUpperBound,
ColorSelected);

                                if(ColorFound){
                                        DrawCapImage();
                                }
                                else{
                                        Application->MessageBoxA("Color not
found!","Error",MB_OK);
                                }
                        break;
                        //Black has been selected
                        case 1:
                                ColorLowerBound[0] = 0;
                                ColorUpperBound[0] = 70;
                                ColorLowerBound[1] = 0;
                                ColorUpperBound[1] = 70;
                                ColorLowerBound[2] = 0;
                                ColorUpperBound[2] = 70;
                                ColorFound = ColorFind(ColorLowerBound, ColorUpperBound,
ColorSelected);
```

```
                                if(ColorFound){
                                        DrawCapImage();
                                }
                                else{
                                        Application->MessageBoxA("Color not
found!","Error",MB_OK);
                                }
                        break;
                        //White has been selected
                        case 2:
                                ColorFound = ColorFind(ColorLowerBound, ColorUpperBound,
ColorSelected);

                                if(ColorFound){
                                        DrawCapImage();
                                }
                                else{
                                        Application->MessageBoxA("Color not
found!","Error",MB_OK);
                                }
                        break;
                }
        }
        else{
                Application->MessageBoxA("Image must be captured first.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------
void __fastcall TCalibrateMenu::CapImageButtonClick(TObject *Sender)
{
        VideoPanel->Visible = false;
```

```
        ImageCirclePanel->Visible = false;
        ColorSelPanel->Visible = true;
        //Clear(void);
        //CamCheck = VideoOCXCal->CaptureToClipboard();
        ColorSelImage->Picture->Bitmap->LoadFromFile(OpenDialog1->FileName);
        CamCheck = true;

        if(CamCheck){
                CapPicture->Assign(Clipboard());
                ImageCapture = true;
                DrawCapImage();
                PixelFormatLabel->Caption = CapPicture->PixelFormat;
                GetRawData(CapPicture);
        }
        else{
                Application->MessageBoxA("Video Capture device is in use or has not
initialized.","Error",MB_OK);
        }
}
//---------------------------------------------------------------------------
//My defined functions
//---------------------------------------------------------------------------
void GetRawData(Graphics::TBitmap *Image){

        BYTE Output[74][74];
        int x, y, T;

        FrameCap.Left=123; FrameCap.Right=197;
        FrameCap.Top=83;  FrameCap.Bottom=157;

        //Capture image raw data and put into primary data storage container
```

```
        for(int y=FrameCap.top; y<FrameCap.bottom; y++){

                ImageRawData = (BYTE *)Image->ScanLine[y];

                        for(int x=FrameCap.left; x<FrameCap.Right; x++){

                                ImageDataMatrixRed[(x-123)][(y-83)] =
ImageRawData[x*3+2];
                                ImageDataMatrixGreen[(x-123)][(y-83)] =
ImageRawData[x*3+1];
                                ImageDataMatrixBlue[(x-123)][(y-83)] = ImageRawData[x*3];
                        }
        }
        //Calculate and perform smoothing transformation for each pixel
        for (y=1; y<73; y++){

                for (x=1; x<73; x++){

                        Output[x][y] = Average(ImageDataMatrixRed, x, y);
                }
        }
        //Calculate and perform threshold transformation of each pixel
        for(y=0; y<74; y++){

                for(x=0; x<74; x++){

                        T = Output[x][y];

                        if(T<=130){
                                ImageDataBW[x][y]=0;
                        }
```

```
                        if(T>130){
                                ImageDataBW[x][y]=255;
                        }
                }
        }
}
//---------------------------------------------------------------------------
void TCalibrateMenu::DrawCapImage(){

        VideoPanel->Visible = false;
        ImageCirclePanel->Visible = false;
        ColorSelPanel->Visible = true;
        ColorSelImage->Picture->Graphic=CapPicture;
        ColorSelImage->Canvas->Pen->Color=clRed;
        ColorSelImage->Canvas->Pen->Mode=pmCopy;
        ColorSelImage->Canvas->Pen->Width=1;

        if(CrossHair){
                points[0]=Point((FarRightx+123),(Firsty+Middley+83));
                points[1]=Point((FarLeftx+123),(Firsty+Middley+83));
                points[2]=Point((FarLeftx+Middlex+123),(Firsty+Middley+83));
                points[3]=Point((FarLeftx+Middlex+123),(Firsty+83));
                points[4]=Point((FarLeftx+Middlex+123),(Bottomy+83));
                points[5]=Point((FarLeftx+Middlex+123),(Firsty+Middley+83));
                ColorSelImage->Canvas->Polygon(points,5);
        }
        CrossHair = false;
}
//---------------------------------------------------------------------------
//Function to find far rigth x
int FarRightX(int m, int n){
```

```
        int i = 0;

        while(i<=10){

                if(ImageDataBW[m][n+1]==0){
                        i = 0;

                        while(ImageDataBW[m][n]==0 && i<=10){
                                n++;
                                i++;
                        }
                }
                m++;
        }
        return INT(m-2);
}
//-------------------------------------------------------------------------
//Function to find far left x
int FarLeftX(int m, int n){

        int i = 0;

        while(i<=10){

                if(ImageDataBW[m][n+1]==0){

                        i = 0;

                        while(ImageDataBW[m][n]==0 && i<=10){
                                n++;
```

```
                                i++;
                        }
                }
                m--;
        }
        Beginy = n;
        return INT(m+2);
}
//-------------------------------------------------------------------------
//Function to find bottom y
int BottomY(int m, int n){

        int i = 0;

        while(i<=10){

                if(ImageDataBW[m][n+1]==0){
                        i = 0;

                        while(ImageDataBW[m][n+1]==0 && i<=10){
                                m++;
                                i++;
                        }
                }
                n++;
        }
        return INT(n-1);
}
//-------------------------------------------------------------------------
//Function to fill white pixels with black if color selected is not right
void BlackFill(int m, int n){
```

```
bool MoveRL = true;
bool MoveLR = false;
bool StuckLR = false;
bool StuckRL = false;
bool RowComplete = false;
int x, y, i = 0;

while(i<=10){

        if(ImageBlkFill[m+1][n+1]==255 && MoveRL){
                while(ImageBlkFill[m+1][n+1]==255||ImageBlkFill[m][n]==255){

                        ImageBlkFill[m][n] = 0;
                        m++;
                        MoveLR = true;
                        MoveRL = false;
                        RowComplete = true;
                        StuckLR = false;
                        i=0;
                }
        }
        else{
                StuckLR = true;
                RowComplete = false;
        }
        if(RowComplete){
                n++;

        }
        if(ImageBlkFill[m-1][n+1]==255 && MoveLR){
```

```
                        while(ImageBlkFill[m-1][n+1]==255||ImageBlkFill[m][n]==255){
                                ImageBlkFill[m][n] = 0;
                                m--;
                                MoveLR = false;
                                MoveRL = true;
                                RowComplete = true;
                                StuckRL = false;
                                i=0;
                        }
                }
                else{
                        StuckRL = true;
                        RowComplete = false;
                }
                if(RowComplete){
                        n++;
                }

                i++;

                if(StuckRL && MoveLR){
                        m--;
                        ImageBlkFill[m][n] = 0;
                }

                if(StuckLR && MoveRL){
                        m++;
                        ImageBlkFill[m][n] = 0;
                }
        }
}
```

```
//-------------------------------------------------------------------------
//Function to find and check for correct color
bool ColorFind(int LowerBnd[3], int UpperBnd[3], int Color){

        int x, y, counter, i, Avg;
        int m, n;
        bool PixelColor[3], ColorFound;


        for(y=0; y<74; y++){

                for(x=0; x<74; x++){
                        ImageBlkFill[x][y] = ImageDataBW[x][y];
                }
        }

        for(int e=0; e<3; e++){
                PixelColor[e] = false;
        }
        //Looking for white pixels
        for(y=0; y<74; y++){

                for(x=0; x<74; x++){
                //Checking to see if white pixels found are connector for horizontal

if((ImageBlkFill[x][y]+ImageBlkFill[x+1][y]+ImageBlkFill[x+2][y])/3 == 255){

                                counter = 0;
                                i = y;
                                //Verifying white pixels are connector for vertical
                                do{
```

```
        counter++;
        i++;
}while(ImageBlkFill[x][i]==255 && i<74);

if(Color==2){
        Firstx = x;
        Firsty = y;
        if(counter>=20){
                FarRightx = FarRightX(Firstx, Firsty);
                FarLeftx = FarLeftX(Firstx, Firsty);
                Bottomy = BottomY(FarLeftx, (Beginy-11));
                CrossHair = true;
                Middlex = (FarRightx - FarLeftx)/2;
                Middley = (Bottomy - Firsty)/2;
                Pixelx = Middlex + FarLeftx;
                Pixely = Middley + Firsty;
                x = 74;
                y = 74;
        }
        else{
                BlackFill(x, y);
                CrossHair = false;
                x = 0;
                y = 0;
        }
}


if(Color==0 || Color==1){
        Firsty = y;
        Firstx = x;
```

```
                if(counter>=5 && counter<20){
                        m = Firstx - 3;
                        n = Firsty - 3;
                        Avg = Average(ImageDataMatrixRed, m, n);

                        if(Avg>=LowerBnd[0] || Avg<=UpperBnd[0]){

                                PixelColor[0] = true;
                        }
                        else{
                                PixelColor[0] = false;
                        }

                        Avg = Average(ImageDataMatrixGreen, m,
n);

                        if(Avg>=LowerBnd[1] && Avg<=UpperBnd[1]){

                                PixelColor[1] = true;
                        }
                        else{
                                PixelColor[1] = false;
                        }

                        Avg = Average(ImageDataMatrixBlue, m, n);

                        if(Avg>=LowerBnd[2] && Avg<=UpperBnd[2]){

                                PixelColor[2] = true;
                        }
                        else{
```

```
                                                        PixelColor[2] = false;
                                                }

if(PixelColor[0]&&PixelColor[1]&&PixelColor[2]){

                                                        FarRightx = FarRightX(Firstx,
Firsty);
                                                        FarLeftx = FarLeftX(Firstx,
Firsty);
                                                        Bottomy = BottomY(FarLeftx,
(Beginy-11));
                                                        CrossHair = true;
                                                        Middlex = (FarRightx -
FarLeftx)/2;
                                                        Middley = (Bottomy - Firsty)/2;
                                                        Pixelx = Middlex + FarLeftx;
                                                        Pixely = Middley + Firsty;
                                                        x = 74;
                                                        y = 74;
                                                }
                                                else{
                                                        BlackFill(Firstx, Firsty);
                                                        CrossHair = false;
                                                        x = 0;
                                                        y = 0;
                                                }
                                        }
                                        else{
                                                BlackFill(Firstx, Firsty);
                                                CrossHair = false;
```

```
                                                x = 0;
                                                y = 0;
                                            }
                                        }
                                    }
                                }
        }
        if(CrossHair){
                ColorFound = true;
                motion->EnVisionChk->Enabled = true;
                //motion->DetectedBut->Enabled = true;
        }
        else{
                ColorFound = false;
        }

        return BOOL(ColorFound);
}
//---------------------------------------------------------------------------
```

LIST OF REFERENCES

1. C.Y. Ho and Jen Sriwattanathamma, *Robot Kinematics Symbolic Automation and Numerical Synthesis*, Ablex Publishing, New Jersey.

2. Antti J. Koivo, *Fundamentals for Control of Robotic Manipulators*, John Wiley & Sons, New York, 1989.

3. Yoram Koren, *Robotics for Engineers*, McGraw Hill, New York.

4. Zhihua Qu, and Darren M. Dawson, *Robust Tracking Control of Robot Manipulators*, IEEE, New York, 1996.

5. Mark W. Spong and M. Vidyasagar, *Robot Dynamics and Control*, John Wiley & Sons, Canada, 1989.

6. Hanqi Zhuang and Zvi S. *Roth, Camera-Aided Robot Calibration*, CRC Press, Florida, 1996.

7. David A. Geller, Programmable Controllers Using the Allen-Bradely SLC-500 Family, Prentice-Hall, Inc., New Jersey, 2000.

8. Arthur R. Weeks, Jr., *Fundamentals of Electronic Image Processing*, SPIE-The International Society for Optical Engineering, Washington, and IEEE Press, New Jersey, 1996.

9. C++ Builder 5 Developer's Guide.

10. Edited by Gaurav Sharma, *Digital Color Imaging Handbook*, CRC Press, Florida, 2003.

11. Earl Gose, Richard Johnsonbauge, and Steve Jost, *Pattern Recognition and Image Analysis*, Prentice Hall, Inc., New Jersey, 1996.